DOCUMENT RESUME

ED 183 895 CE 024 525

TITLE Military Curricula for Vocational & Technical Education. Communications Computer Programmer, 4-2.
INSTITUTION Air Force Training Command, Keesler AFB, Miss.; Ohio State Univ., Columbus. National Center for Research in Vocational Education.
SPONS AGENCY Bureau of Occupational and Adult Education (DHEW/OE), Washington, D.C.
PUB DATE Aug 78
NOTE 689p.; Not available in paper copy due to small type.

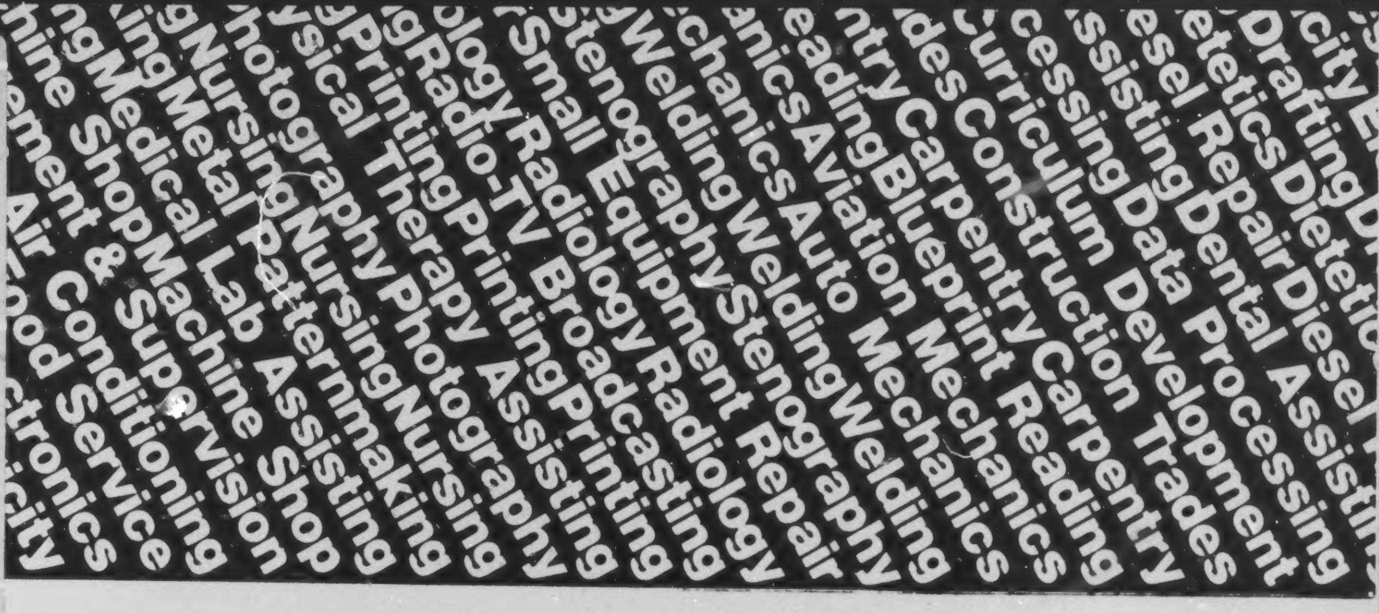
EDRS PRICE MF04 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS Autoinstructional Aids; Behavioral Objectives; *Computer Science; *Computer Science Education; Course Descriptions; Curriculum Guides; Data Processing; High Schools; Learning Activities; Postsecondary Education; Problem Solving; Programed Instructional Materials; *Programing; *Programing Languages; Study Guides; Technical Mathematics; Workbooks
IDENTIFIERS FORTRAN Programing Language; Military Curriculum Project

ABSTRACT

These student materials--study guides, handouts (some are manuals), a workbook, and programmed texts--for a secondary-postsecondary-level course for communications computer programmer are one of a number of military-developed curriculum packages selected for adaptation to vocational instruction and curriculum development in a civilian setting. A plan of instruction, which suggests number of hours of class time devoted to each course objective and support material and guidance, covers these topics: computer mathematics, data representation, computer logic functions, and logical development of problem solving. Contents of the unit, Computer Programming Principles, include a handout with student notes, a study guide with learning materials, and a workbook with classroom and homework exercises. A programmed text is provided for the unit, Computer Logic Functions. The unit, Top Down Structured Programming, is a self-instruction text. The unit on Programming Principles is a study guide with learning materials. Student manuals (handouts) comprise the units, Fortran Language and Examples of Structured Code. (YLB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED183895



Military Curricula for Vocational & Technical Education

COMMUNICATIONS COMPUTER PROGRAMMER

4-2

CE 024 525

DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THE NATIONAL CENTER
FOR RESEARCH IN VOCATIONAL EDUCATION
THE OHIO STATE UNIVERSITY

This military technical training course has been selected and adapted by The Center for Vocational Education for "Trial Implementation of a Model System to Provide Military Curriculum Materials for Use in Vocational and Technical Education," a project sponsored by the Bureau of Occupational and Adult Education, U.S. Department of Health, Education, and Welfare.

MILITARY CURRICULUM MATERIALS

The military-developed curriculum materials in this course package were selected by the National Center for Research in Vocational Education Military Curriculum Project for dissemination to the six regional Curriculum Coordination Centers and other instructional materials agencies. The purpose of disseminating these courses was to make curriculum materials developed by the military more accessible to vocational educators in the civilian setting.

The course materials were acquired, evaluated by project staff and practitioners in the field, and prepared for dissemination. Materials which were specific to the military were deleted, copyrighted materials were either omitted or approval for their use was obtained. These course packages contain curriculum resource materials which can be adapted to support vocational instruction and curriculum development.

The National Center Mission Statement

The National Center for Research in Vocational Education's mission is to increase the ability of diverse agencies, institutions, and organizations to solve educational problems relating to individual career planning, preparation, and progression. The National Center fulfills its mission by:

- Generating knowledge through research
- Developing educational programs and products
- Evaluating individual program needs and outcomes
- Installing educational programs and products
- Operating information systems and services
- Conducting leadership development and training programs

FOR FURTHER INFORMATION ABOUT
Military Curriculum Materials
WRITE OR CALL
Program Information Office
The National Center for Research in Vocational Education
The Ohio State University
1960 Kenny Road, Columbus, Ohio 43210
Telephone: 614/486-3655 or Toll Free 800/848-4815 within the continental U.S. (except Ohio)

THE NATIONAL CENTER
FOR RESEARCH IN VOCATIONAL EDUCATION
The Ohio State University · 1960 Kenny Road · Columbus, Ohio 43210
Tel. (614) 486-3655 Cable: CTVOCEDOSU/Columbus, Ohio

Military Curriculum Materials for Vocational and Technical Education

Information and Field
Services Division

The National Center for Research
in Vocational Education

Military Curriculum Materials Dissemination Is . . .

an activity to increase the accessibility of military-developed curriculum materials to vocational and technical educators.

This project, funded by the U.S. Office of Education, includes the identification and acquisition of curriculum materials in print form from the Coast Guard, Air Force, Army, Marine Corps and Navy.

Access to military curriculum materials is provided through a "Joint Memorandum of Understanding" between the U.S. Office of Education and the Department of Defense.

The acquired materials are reviewed by staff and subject matter specialists, and courses deemed applicable to vocational and technical education are selected for dissemination.

The National Center for Research in Vocational Education is the U.S. Office of Education's designated representative to acquire the materials and conduct the project activities.

Project Staff:

Wesley E. Budke, Ph.D., Director
National Center Clearinghouse

Shirley A. Chase, Ph.D.
Project Director

What Materials Are Available?

One hundred twenty courses on microfiche (thirteen in paper form) and descriptions of each have been provided to the vocational Curriculum Coordination Centers and other instructional materials agencies for dissemination.

Course materials include programmed instruction, curriculum outlines, instructor guides, student workbooks and technical manuals.

The 120 courses represent the following sixteen vocational subject areas:

- Agriculture
- Aviation
- Building & Construction Trades
- Clerical Occupations
- Communications
- Drafting
- Electronics
- Engine Mechanics
- Food Service
- Health
- Heating & Air Conditioning
- Machine Shop
- Management & Supervision
- Meteorology & Navigation
- Photography
- Public Service

The number of courses and the subject areas represented will expand as additional materials with application to vocational and technical education are identified and selected for dissemination.

How Can These Materials Be Obtained?

Contact the Curriculum Coordination Center in your region for information on obtaining materials (e.g., availability and cost). They will respond to your request directly or refer you to an instructional materials agency closer to you.

CURRICULUM COORDINATION CENTERS

EAST CENTRAL
Rebecca S. Douglass
Director
100 North First Street
Springfield, IL 62777
217/782-0759

MIDWEST
Robert Patton
Director
1515 West Sixth Ave.
Stillwater, OK 74704
405/377-2000

NORTHEAST
Joseph F. Kelly, Ph.D.
Director
225 West State Street
Trenton, NJ 08625
609/292-6562

NORTHWEST
William Daniels
Director
Building 17
Airdustrial Park
Olympia, WA 98504
206/753-0879

SOUTHEAST
James F. Shill, Ph.D.
Director
Mississippi State University
Drawer DX
Mississippi State, MS 39762
601/325-2510

WESTERN
Lawrence F. H. Zane, Ph.D.
Director
1776 University Ave.
Honolulu, HI 96822
808/948-7834

COMMUNICATIONS COMPUTER PROGRAMMER

3ABR51131

Classroom Course

Developed by:

United States Air Force

Development and Review Dates:

August 1978

Occupational Area:

Target Audiences:

11 - Adult

Print Pages: 695

Microfiche: 12

Availability:

Vocational Curriculum Coordination Centers

Contents:	**Type of Materials:**						**Instructional Design:**				**Type of Instruction:**		
	Lesson Plans:	Programmed Text:	Student Workbook:	Handouts:	Text Materials:	Audio-Visuals:	Performance Objectives:	Tests:	Review Exercises:	Additional Materials Required:	Group Instruction:	Individualized:	
Block I													
Programming													
Principles	●	●	●	●	●	x	●				●	●	

X Materials are recommended but not provided.

Course Description:

This course includes the following topics:

Computer Mathematics (12.5 hours)
Data Representation (5 hours)
Computer Logic Functions (5 hours)
Logical Development of Problem Solutions (65 hours)

Student materials include handouts, study guides, a workbook, and programmed texts.

Instruction materials include a Plan of Instruction and transparency masters.

COMMUNICATIONS COMPUTER PROGRAMMER

Table of Contents

	Page
Plan of Instruction	2
Computer Programming Principles:	
Student Handout	30
Study Guide	97
Workbook	190
Computer Logic Functions - Programmed Text	306
Top Down Structured Programming - Self Instruction	358
Programming Principles - Study Guide	519
Fortran Language - Student Handout	589
Examples of Structured Code - Student Handout	665

1

MODIFICATIONS

P.O.I. pp. 1-14 of this publication has (have) been deleted in adapting this material for use in Vocational and Technical Education. Deleted material involves extensive use of military forms, procedures, systems, etc. and was not considered appropriate for use in vocational and technical education.

2

PLAN OF INSTRUCTION/LESSON PLAN PART I

NAME OF INSTRUCTOR	COURSE TITLE Communications Computer Programmer
BLOCK NUMBER I	BLOCK TITLE Programming Principles

1 COURSE CONTENT

6. Project Management

a. Given a completed Program Evaluation Review Technique (PERT) network and a two column listing, match the items in column two with the most appropriate item in column one to an accuracy of at least 75%. Each item in column two may be used once, more than once, or not at all.
CTS: 1a Meas: W

(1) History of PERT

(2) Concepts of PERT

SUPPORT MATERIALS AND GUIDANCE

Student Instructional Materials
HO KDA-472, Project Management (PERT)

Training Methods
Lecture/Discussion

SUPERVISOR APPROVAL OF LESSON PLAN (PART II)

SIGNATURE AND DATE	SIGNATURE AND DATE

PLAN OF INSTRUCTION NUMBER	DATE	PAGE NO.
E30ZR3024D 000	16 August 1978	13

NAME OF INSTRUCTOR	COURSE TITLE
	Communications Computer Programmer

BLOCK NUMBER	BLOCK TITLE
I	Programming Principles

1 COURSE CONTENT	2 TIME

7. Computer Mathematics

a. Given a series of numbers; understand the procedures and perform conversions, including all combinations; and addition, subtraction, and complementation of binary, octal, decimal, and hexadecimal numbers.
CTS: 1b,1c Meas: W, PT

- (1) Rules of algebra
- (2) Number systems
  - (a) Decimal
  - (b) Octal
  - (c) Binary
  - (d) Hexadecimal
- (3) Addition, subtraction, and complements
  - (a) Decimal
  - (b) Octal
  - (c) Binary
  - (d) Hexadecimal

SUPERVISOR APPROVAL OF LESSON PLAN (PART II)

SIGNATURE AND DATE	SIGNATURE AND DATE

PLAN OF INSTRUCTION NUMBER	DATE	PAGE NO.
E30ZR3024D 000	16 August 1978	15

ATC FORM OCT 75 133 PREVIOUS EDITION IS OBSOLETE ATC Keesler 8-1856

SUPPORT MATERIALS AND GUIDANCE

Student Instructional Materials

HO KDA-469, Computer Programming Principles Student Notes
ST KDA-470, Computer Programming Principles
WB KDA-471, Computer Programming Principles

Audio Visual Aids

Transparencies, Math

Training Methods

Discussion
Performance

NAME OF INSTRUCTOR	COURSE TITLE Communications Computer Programmer

BLOCK NUMBER I	BLOCK TITLE Programming Principles

1 COURSE CONTENT	2 TIME

8. Data Representation

a. Given a group of definitions, identify the appropriate terms.
CTS: 1a Meas: W

- (1) External
  - (a) Bit
  - (b) Byte
  - (c) Character
  - (d) Word
  - (e) Field
  - (f) Record
  - (g) File
- (2) Internal
  - (a) ASCII
  - (b) BCD
  - (c) Integer
  - (d) Floating point

b. Given all applicable references, the contents (in Octal) of several computer locations, and the mode the information is stored in, select the value that correctly represents the stored data.
CTS: 1a Meas: W

- (1) Numeric data
- (2) Alpha-numeric data

SUPERVISOR APPROVAL OF LESSON PLAN (PART II)

SIGNATURE AND DATE	SIGNATURE AND DATE

PLAN OF INSTRUCTION NUMBER	DATE	PAGE NO.
E30ZR3024D 000	16 August 1978	17

SUPPORT MATERIALS AND GUIDANCE

Student Instructional Materials
HO KDA-469
ST KDA-470
WB KDA-471

Audio Visual Aids
Transparencies, Data Respresentation

Training Methods
Discussion/Demonstration

7.

NAME OF INSTRUCTOR	COURSE TITLE Communications Computer Programmer
BLOCK NUMBER I	BLOCK TITLE Programming Principles

1 COURSE CONTENT | 2 TIME

9. Computer Logic Functions

a. Given a series of problems and a truth table of logic functions, name the logic function and determine the logic function results using binary numbers.
CTS: 1d Meas: W, PT

(1) Logic functions

(a) OR/AND

(b) Exclusive OR

(c) NAND/NOR

(d) NOT

SUPPORT MATERIALS AND GUIDANCE

Student Instructional Materials
PT KDA-305, Computer Logic Functions

Audio Visual Aids
Transparencies, Logic

Training Methods
Discussion

Instructional Guidance
Guide student performance using KDA-305.

SUPERVISOR APPROVAL OF LESSON PLAN (PART II)

SIGNATURE AND DATE	SIGNATURE AND DATE

PLAN OF INSTRUCTION NUMBER	DATE	PAGE NO.
E30ZR3024D 000	16 August 1978	19

ATC FORM OCT 75 133 PREVIOUS EDITION IS OBSOLETE ATC Keesler 8-1856

8

PLAN OF INSTRUCTION/LESSON PLAN PART I

NAME OF INSTRUCTOR	COURSE TITLE Communications Computer Programmer

BLOCK NUMBER I	BLOCK TITLE Programming Principles

1 COURSE CONTENT	2 TIME

10. Logical Development of Problem Solutions

a. Given a narrative description of problems requiring straight line, branching, and looping flowcharts, analyze and construct flowcharts to show required operations.
CTS: 1e,1g(1),(2),(3),(4) Meas: PT

(1) Purpose of flowcharts

(a) Aid in coding

(b) Debugging

(c) Documentation

(2) Flowchart symbols and logical operators

(a) Terminal

(b) Processing

(c) Decision

(d) Input/output

(e) Subroutines

(f) Connectors

b. Given all available reference materials and a set of problem specifications; use the Top Down concept of problem solving to draw a block diagram, write a narrative description of the block diagram, and draw a structured flow chart showing the steps necessary to solve the problem.
CTS: 1e,1f,1g(1),(2),(3),(4) Meas: W, PT

(1) Introduction to Top Down Design and Structured Programming (TDSP)

(a) Purpose of TDSP

SUPERVISOR APPROVAL OF LESSON PLAN (PART II)

SIGNATURE AND DATE	SIGNATURE AND DATE

PLAN OF INSTRUCTION NUMBER	DATE	PAGE NO.
E3OZR3024D 000	16 August 1978	21

ATC FORM OCT 75 133 PREVIOUS EDITION IS OBSOLETE

(b) Governing directives

(c) Top Down Design

(d) Top Down Documentation

(e) Structured walkthroughs

(f) Top Down implementation

(g) Programming teams

(i) Program Design Language (PDL)

(2) Define the problem (HIPO)

(a) Use of Hierarchy Block Diagram

(b) Narrative description of input, process, and output

(c) Identification of needed information

(3) Methods of problem solving

(a) Direct

(b) Enumerating

(c) Scientific trial and error

(d) Simulation

c. Given all available reference materials and a set of problem specifications; use the Top Down concept of problem solving to draw a block diagram, write a narrative description of the block diagram, and draw a structured flow chart showing the steps necessary to solve the problem.
CTS: 1e,1f,1g(1),(2),(3),(4) Meas: W, PT

(1) Flowchart the solution

(a) Definition of a function

(b) Flowchart highest level functions first

(c) Flowchart symbols

1 Sequence

2 If ... then ... else ...

3 Do while

4 Do until

5 Case, select, switch

6 Loop, exitif, endloop

d. Given all available reference materials and a set of problem specifications, construct a HIPO/flowchart to solve the problem(s). CTS: 1e,1f,1g(1),(2),(3),(4) Meas: W, PT

(1) Search

(2) Sort

(3) Insertion

(4) Deletion

(5) Merge

e. Given a flowchart and a description of data with preassigned valves, analyze the flowchart and determine the values of various items at selected points within the flowchart. CTS: 1e,1g(1),(2),(3),(4) Meas: PT

f. Conduct a structured walkthrough utilizing a flowchart and HIPO documentation. CTS: 1f Meas: PT

21

11

SUPPORT MATERIALS AND GUIDANCE

Student Instructional Materials
ST KDA-295, Top Down Structured Programming
HO KDA-469
ST KDA-470
WB KDA-471
SG KDA-478, Programming Principles

Audio Visual Aids
Transparencies, Flowchart Structure
Transparencies, Flowchart Symbols

Training Methods
Lecture/Discussion
Performance

MODIFICATIONS

Pages 13 - 29 of this course have been deleted due to copyrighted material.

STUDENT HANDOUT
3ABR51131B
KDA-469

30

Technical Training

Programming Specialist (Honeywell)

COMPUTER PROGRAMMING PRINCIPLES
STUDENT NOTES

July 1976

USAF SCHOOL OF APPLIED AEROSPACE SCIENCES
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler 7-1103

DO NOT USE ON THE JOB

24

INTRODUCTION Pg 1-4
DIGITAL COMPUTER ELEMENTS 5-11
LANGUAGE LEVELS 12-14
HISTORY 15-30
COMPUTER GENERATIONS 31-40
H6000 HARDWARE 41-47
MATH 48-55
LOGIC 56-58
STRUCTURED FLOWCHART 59-61
FLOWCHART TYPES 62-65

DEFINITION

A COMPUTER IS ACTUALLY A DATA PROCESSING DEVICE THAT PERFORMS MATHEMATICAL AND LOGICAL OPERATIONS ON DATA IN A PREARRANGED AND CONTROLLED MANNER.

1

COMPUTER FUNCTIONS

ARITHMETIC

A + B = C

(2) (3) (5)

LOGICAL

A > B STATUS INDICATOR

(1) (2) 0 (NO)

2

TERMINOLOGY

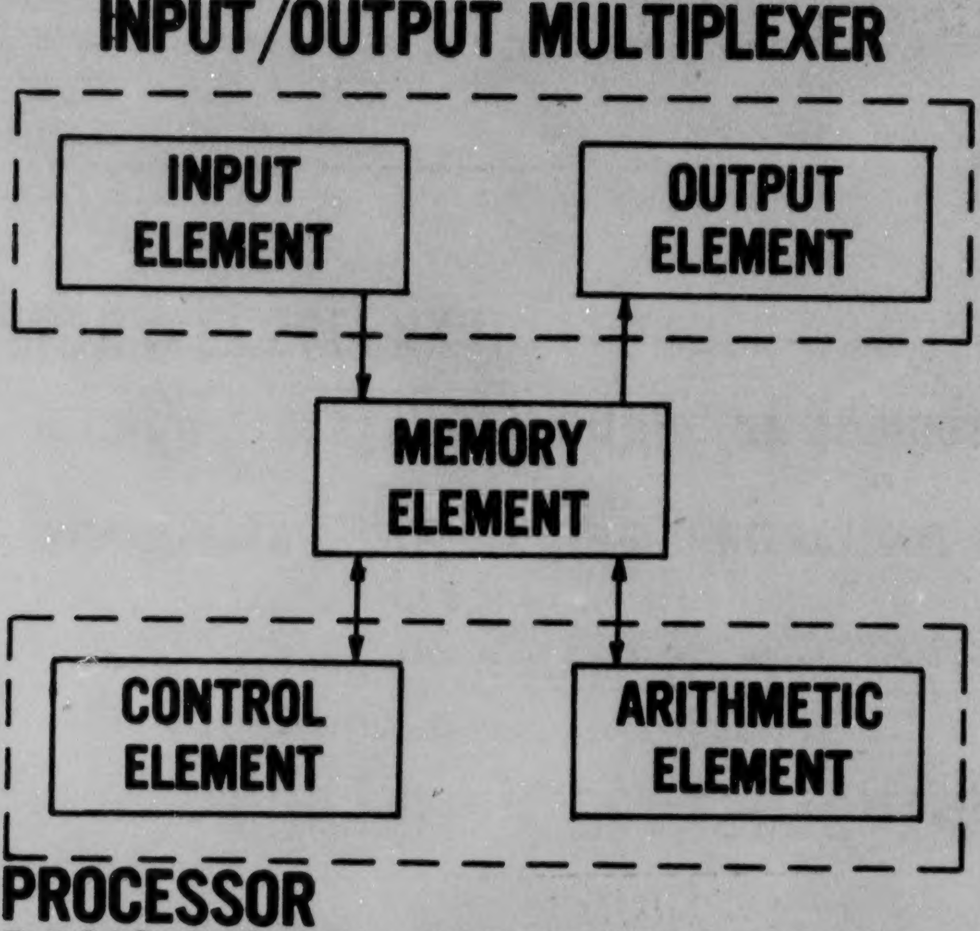
- BYTE
- BIT
- PROGRAM
- INSTRUCTION
- ADDRESS
- RECORD
- RADIX
- FILE
- I/O
- TAPE
- DISK
- REGISTER
- SUBROUTINE

COMPUTER CLASSIFICATION

ANALOG COMPUTER
DIGITAL COMPUTER
HYBRID COMPUTER
SPECIAL PURPOSE COMPUTER
GENERAL PURPOSE COMPUTER

4

36



BASIC DIGITAL COMPUTER ELEMENTS

BASIC DIGITAL COMPUTER ELEMENTS

INPUT ELEMENT

Buffer between external input devices and internal computer elements. It converts symbolic source language code into machine usable binary code.

6

38

MEMORY ELEMENT

A large storage area within the computer where data (both program instructions and program data) can be stored and accessed.

7

CONTROL ELEMENT

Interprets and carries out the instructions of a program. This includes fetching instructions from memory, decoding each instruction, and applying the proper signals to the arithmetic element and other registers.

ARITHMETIC ELEMENT

A series of buffers (accumulators) and registers use to hold data while it is being acted on by the logic and/or arithmetic circuits.

40

OUTPUT ELEMENT

Functions to compensate for the different rate in data transfer of the external output devices and the internal computer elements.

9

INPUT DEVICES

- Card Reader
- Paper Tape
- Magnetic Tape
- Disk, Drum
- Console/Terminal
- Another Computer
- Communications Line
- Optical Scanner

OUTPUT DEVICES

- Line Printer
- Card Punch
- Plotter
- Console/Terminal
- Another Computer
- Paper Tape
- Magnetic Tape
- Disk, Drum

LEVELS OF PROGRAMMING LANGUAGES

MACHINE LANGUAGE

ASSEMBLY LANGUAGE

COMPILER LANGUAGE

MACHINE LANGUAGE

Is basically the lowest level of languages that can be programmed. Each machine language instruction has an equivalent hardware circuit to perform the specified operation .

ASSEMBLY LANGUAGE

Converts a program, written in mnemonic symbols, into a machine language program. One assembler instruction translates to one machine language instruction.

COMPILER LANGUAGE

Converts a program, written in symbolic coding, into a machine language program. One compiler instruction translates to several machine language instructions.

14

46

ERAS

- MANUAL
- MECHANICAL
- PCAM
- ELECTRONIC

15

MANUAL ERA

- "REMINDERS"
  - STICKS
  - STONES
  - FINGERS
- ABACUS

  3000 BC

48

MECHANICAL ERA

- PASCAL
- LEIBNITZ
- JACQUARD
- BABBAGE

17

BLAISE PASCAL

FRENCH MATHEMATICIAN

1st ADDING MACHINE - 1642

- SERIES OF WHEELS : 0 to 9
- "CARRY" LEVER AT 9 + 1 (10)
- LIMITED TO ADD OR SUBTRACT

50

GOTTFRIED WILHELM VON LEIBNITZ

GERMAN MATHEMATICIAN

''FOUR FUNCTION'' MACHINE - 1673

+ STEPPED
- UNSTEPPED
× SERIES OF ADDS
÷ SERIES OF SUBTRACTS

JACQUARD

CONTROLLED WEAVING ON LOOM - 1801

- USED A NOTCHED CARD
- PUBLIC FEAR OF MACHINES LIMITED ITS USE

20

CHARLES BABBAGE

ENGLISH MATHEMATICIAN

"DIFFERENCE" MACHINE - 1822

- CONTROLLED BY PUNCHED CARDS
- CAPABLE OF MAKING LOGICAL DECISIONS

"ANALYTICAL ENGINE" - 1833

- TOO ADVANCED FOR CURRENT TECHNOLOGY
- 100 YEARS WHEN HIS PRINCIPLES DEVELOPED

PCAM ERA

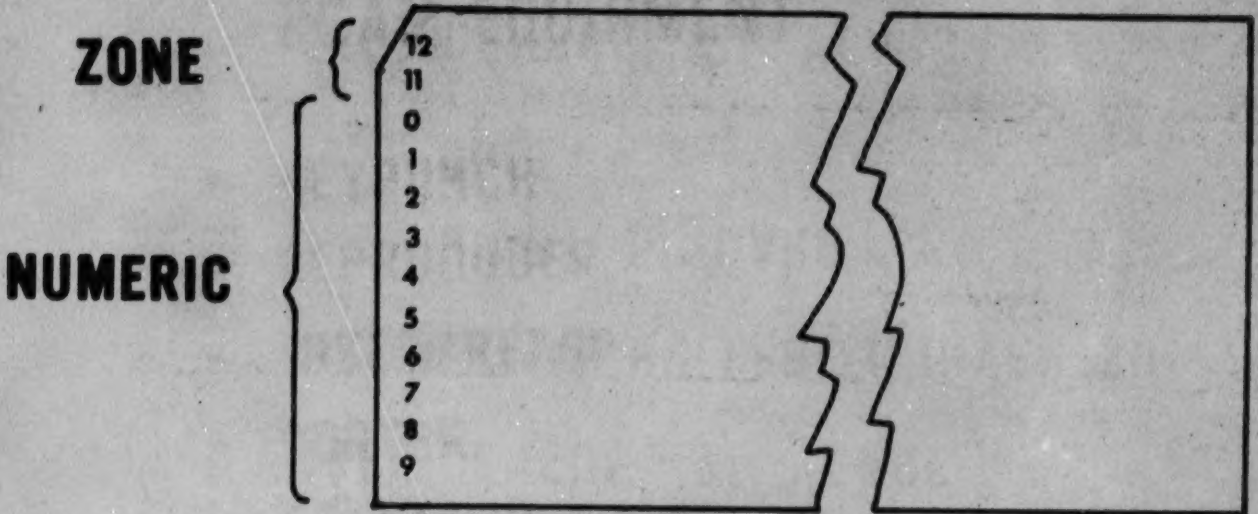
1880 CENSUS - 7½ YEARS

DR. HERMAN HOLLERITH INVENTED:

- 3" BY 5" CARD
- CODE

1890 CENSUS - 2½ YEARS

HOLLERITH CODE



A→I : 12 + 1-9
J→R : 11 + 1-9
S→Z : 0 + 2-9

DR HOLLERITH

FORMED:

1896 - TABULATING MACHINE COMPANY

1911 - MERGED WITH

INT'L TIME RECORDING CO., AND
DAYTON SCALE CO., TO FORM
COMPUTING - TABULATING - RECORDING CO.

1924 - CHANGED NAME TO "IBM"

24

56

PCAM EQUIPMENT

- KEYPUNCH
- REPRODUCER
- INTERPRETER
- SORTER
- COLLATOR
- ACCOUNTING MACHINES

25

ELECTRONIC ERA

- AIKEN
- MAUCHLY
- ECKERT

MARK I - 1944

- PROF AIKEN - HARVARD
- AUTOMATIC SEQUENCE CONTROLLED CALCULATOR
- USED RELAYS AND SWITCHES

27

5

ENIAC - 1945

- MAUCHLY AND ECKERT
- ELECTRONIC NUMERICAL INTEGRATOR AND CALCULATOR
  - 18000 VACUUM TUBES
  - CALCULATE BALLISTICS AND AERONAUTICS
  - 1st TRULY ELECTRONIC COMPUTER

60

EDVAC - 1952

- MAUCHLY AND ECKERT
- ELECTRONIC DISCRETE VARIABLE AUTOMATIC COMPUTER

  *ONLY 3500 VACUUM TUBES

  *STORED PROGRAM

MAUCHLY AND ECKERT

1946 - ELECTRONIC CONTROL COMPANY

1951 - UNIVERSAL AUTOMATIC COMPUTER
(UNIVAC I)

*1951 - BUREAU OF CENSUS

30

CONCEPTS

THIRD GENERATION

SECOND GENERATION

FIRST GENERATION

PCAM

GENERATIONS OF COMPUTERS

- 1st 1946 → 1958
- 2nd 1958 → 1965
- 3rd 1965 → 1973???
- 4th 1970 → ????

64

1ST GENERATION - 1946→1958

- VACUUM TUBES -

FEATURES:

- HIGH HEAT
- HUGE TUBES (16" LONG)
- SMALL MEMORIES
- PAPER TAPE/PUNCHED CARD INPUT
- SLOW
- MACHINE LANGUAGE
- EXPENSIVE
- NO O.S.

6

2ND GENERATION - 1958→1965

- TRANSISTORS -

- MAGNETIC TAPE -

FEATURES:

- LESS HEAT
- SMALLER
- FASTER
- MORE RELIABLE
- LESS POWER REQUIRED
- "SYMBOLIC" LANGUAGE
- OFF-LINE STORAGE
- STILL SEQUENTIAL
- MAGNETIC CORE
- SIMPLE O.S.

66

3RD GENERATION - 1965-1973 ???

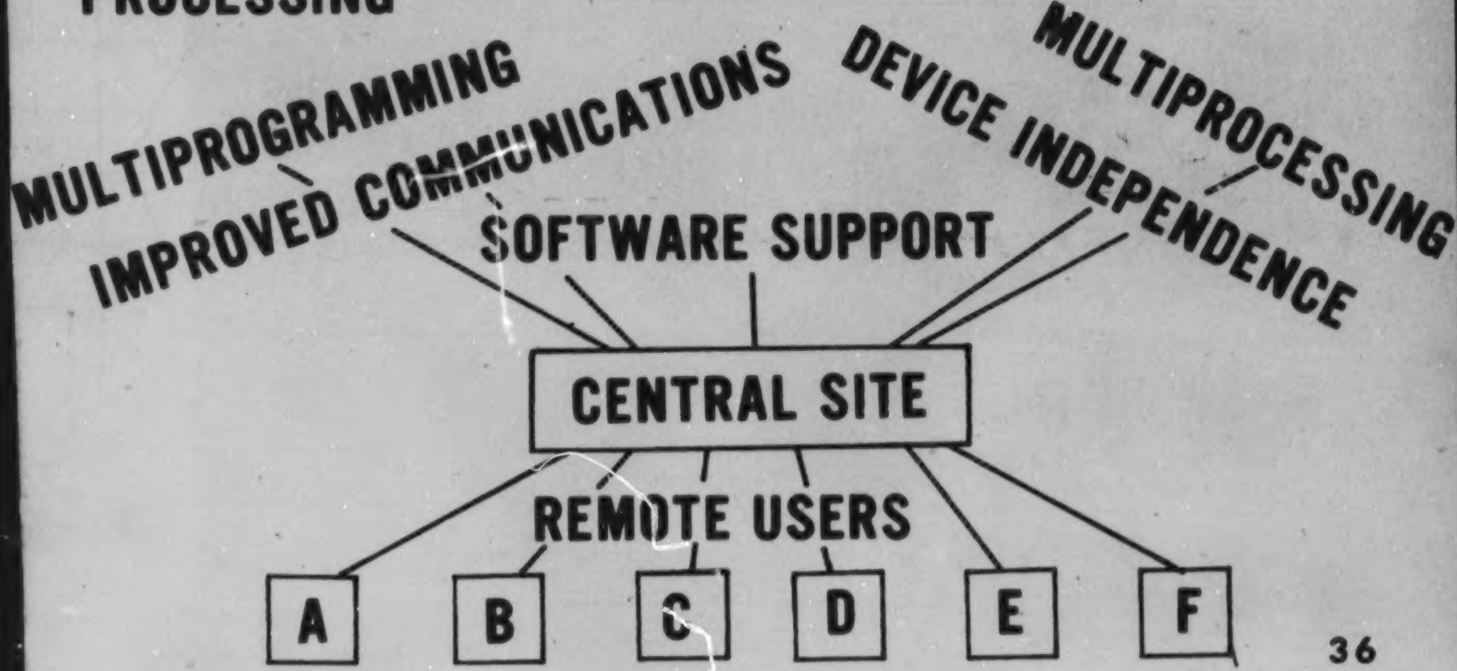
- INTEGRATED CIRCUITS -

FEATURES:

- SOLID STATE
- LOWER COST
- SMALLER SIZE
- LARGE MEMORY
- INTERRUPT CAPABILITY
- REMOTE PROCESSING
- DATA BASE TECHNOLOGY - IDS, ISP, DMS
- MULTIPROCESSING/MULTIPROGRAMMING
- MASS STORAGE VS TAPE
- FASTER, MORE RELIABLE

3rd GENERATION - REMOTE PROCESSING -

THIRD GENERATION ADVANCES HAVE INITIATED CAPABILITY FOR TERMINAL AND REMOTE SITE PROCESSING



MULTIPROGRAMMING

IMPROVED COMMUNICATIONS

SOFTWARE SUPPORT

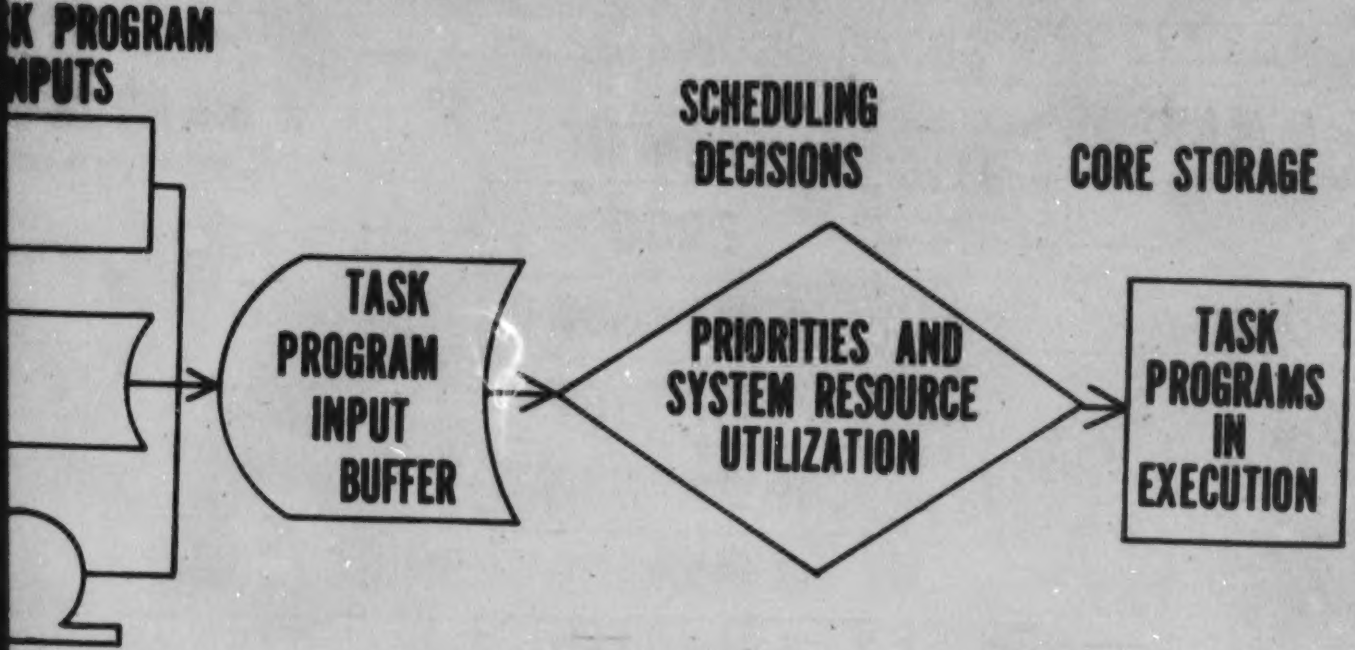
DEVICE INDEPENDENCE

MULTIPROCESSING

CENTRAL SITE

REMOTE USERS

3RD GENERATION SCHEDULING



K PROGRAM
NPUTS

SCHEDULING
DECISIONS

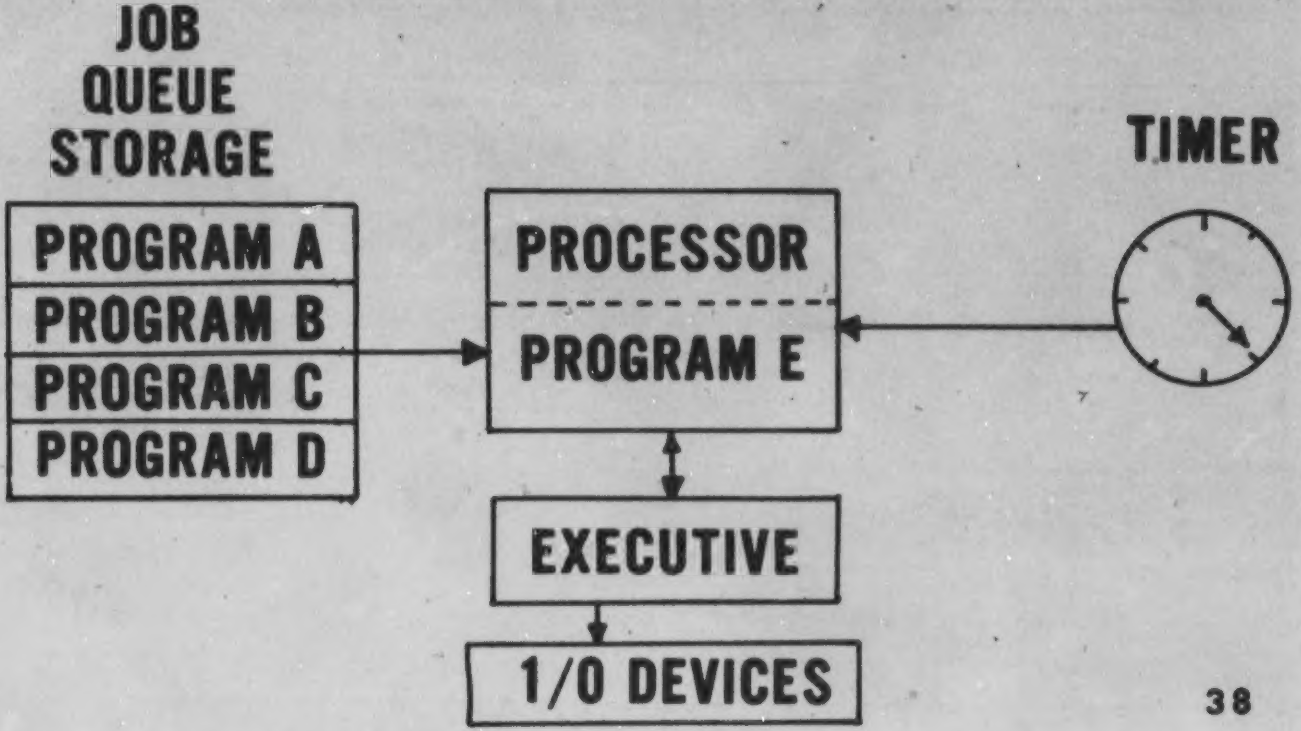
CORE STORAGE

TASK
PROGRAM
INPUT
BUFFER

PRIORITIES AND
SYSTEM RESOURCE
UTILIZATION

TASK
PROGRAMS
IN
EXECUTION

MULTIPROGRAMMING
3RD GENERATION CONCEPT



JOB
QUEUE
STORAGE

PROGRAM A
PROGRAM B
PROGRAM C
PROGRAM D

PROCESSOR

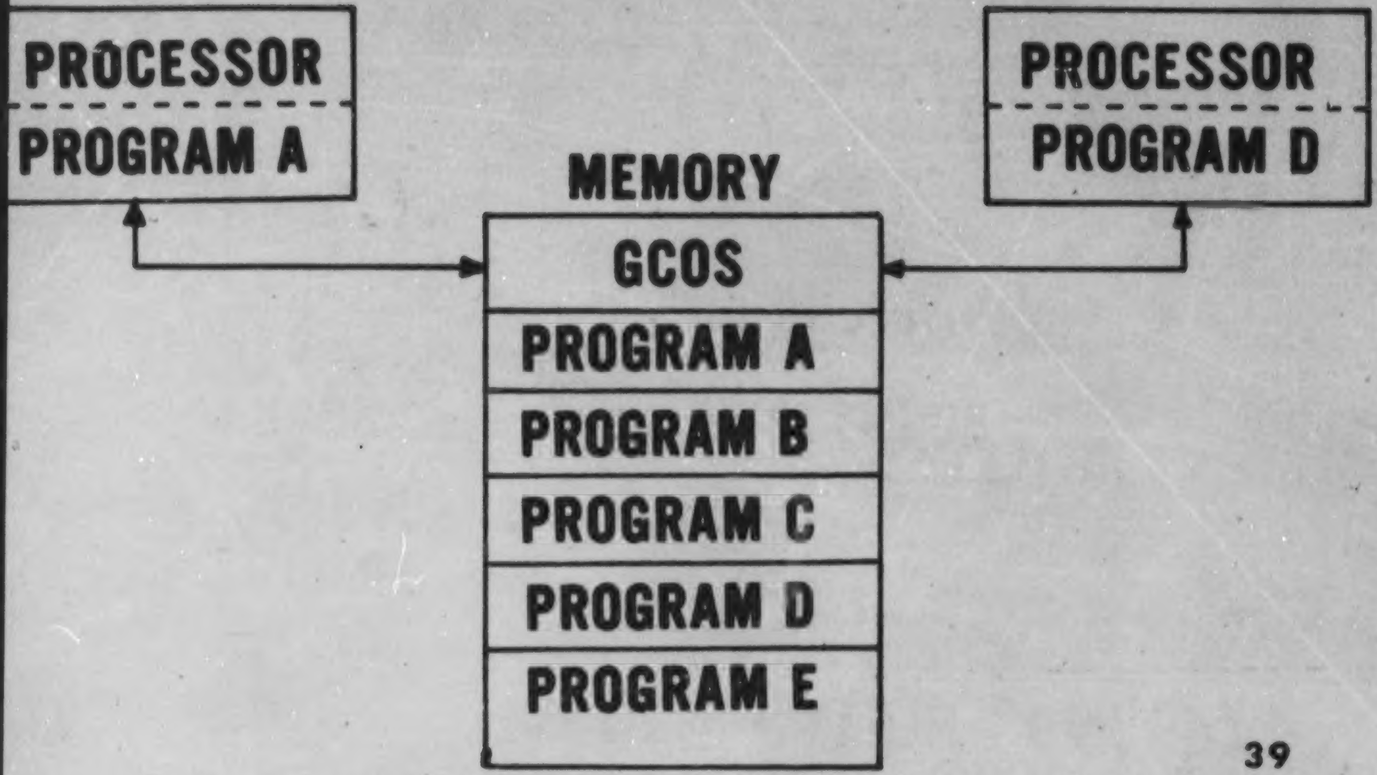
PROGRAM E

TIMER

EXECUTIVE

1/0 DEVICES

MULTIPROCESSING CONCEPT



PROCESSOR

PROGRAM A

MEMORY

GCOS

PROGRAM A

PROGRAM B

PROGRAM C

PROGRAM D

PROGRAM E

PROCESSOR

PROGRAM D

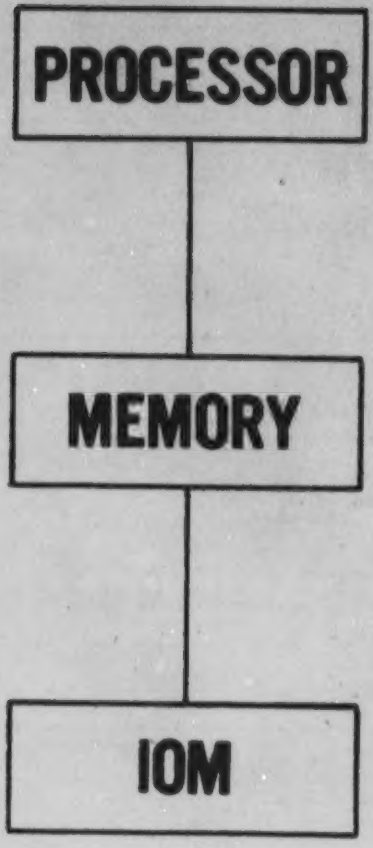
7

4TH GENERATION - 1970→????

- IN RETROSPECT -

NOMINEES:

- SPEED ... 10^{-9} & UP
- ROM
- FILM FOR MEMORY
- I/O DEVICES OCR, AUDIO, SCAN HANDWRITING
- SIZE VS CAPABILITY

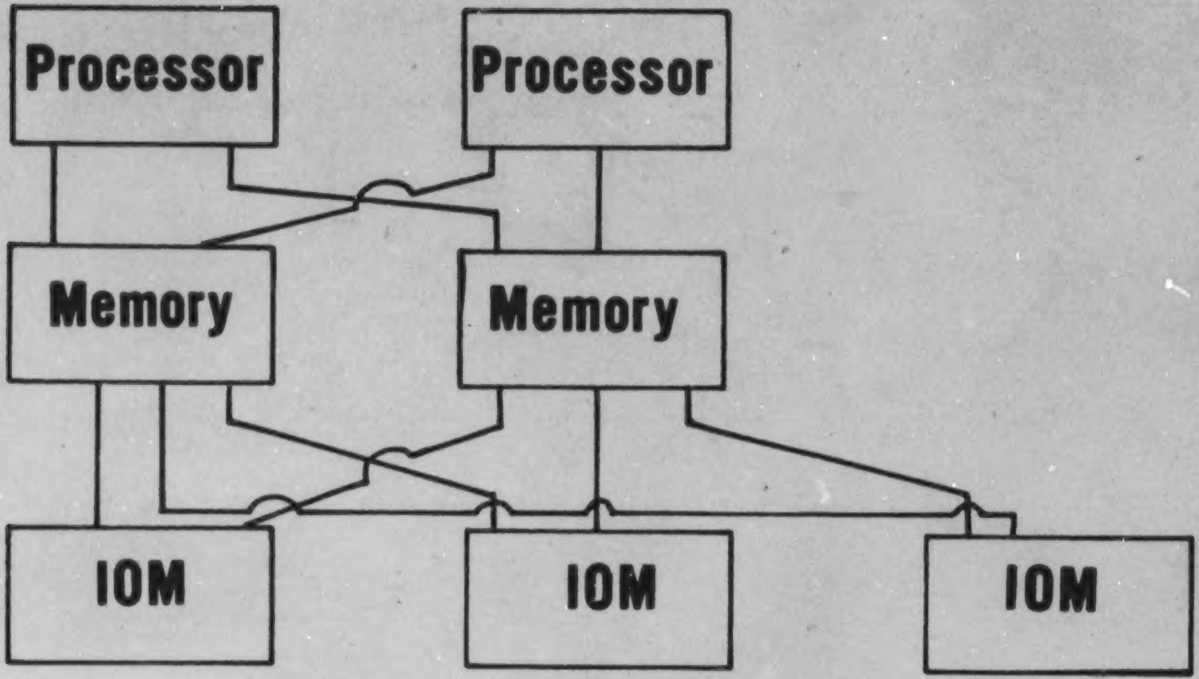


PROCESSOR

MEMORY

IOM

CENTRAL UNIT OF H6000
(ASYNCHRONOUS I/O)



Processor

Processor

Memory

Memory

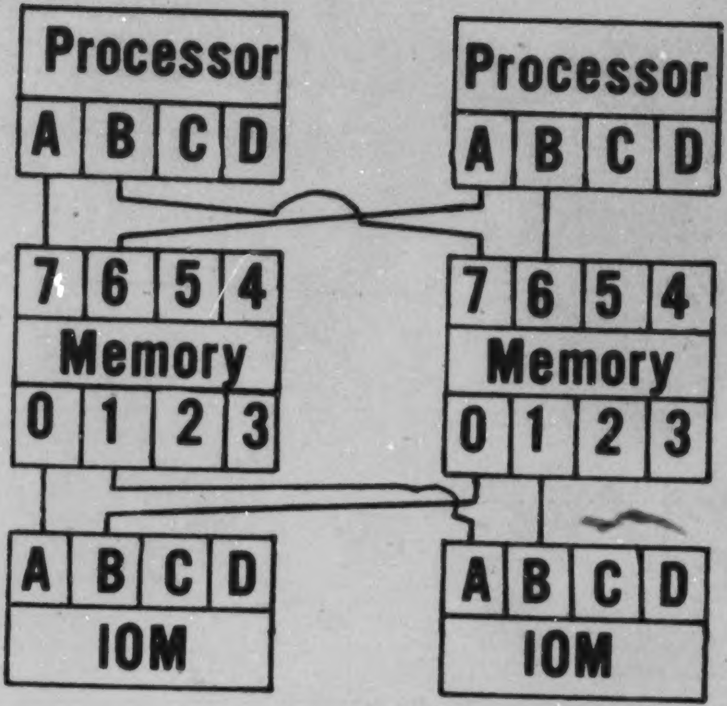
IOM

IOM

IOM

FUNCTIONAL MODULARITY

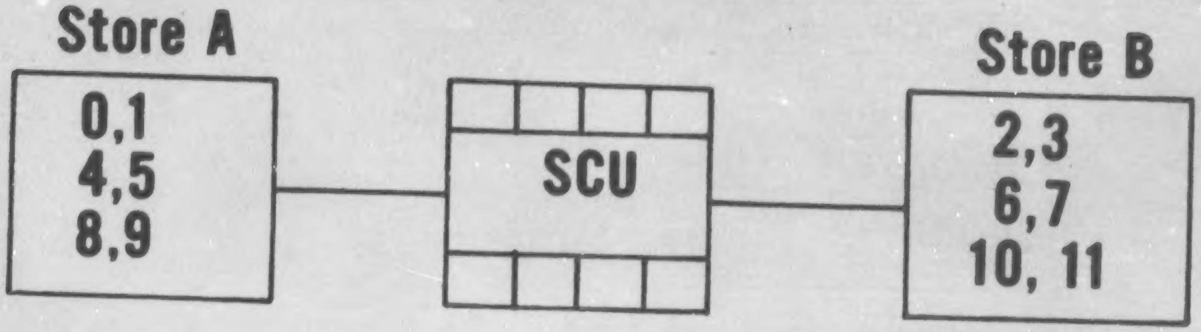
74



PORT CONNECTIONS

43

MEMORY MODULE



Store A

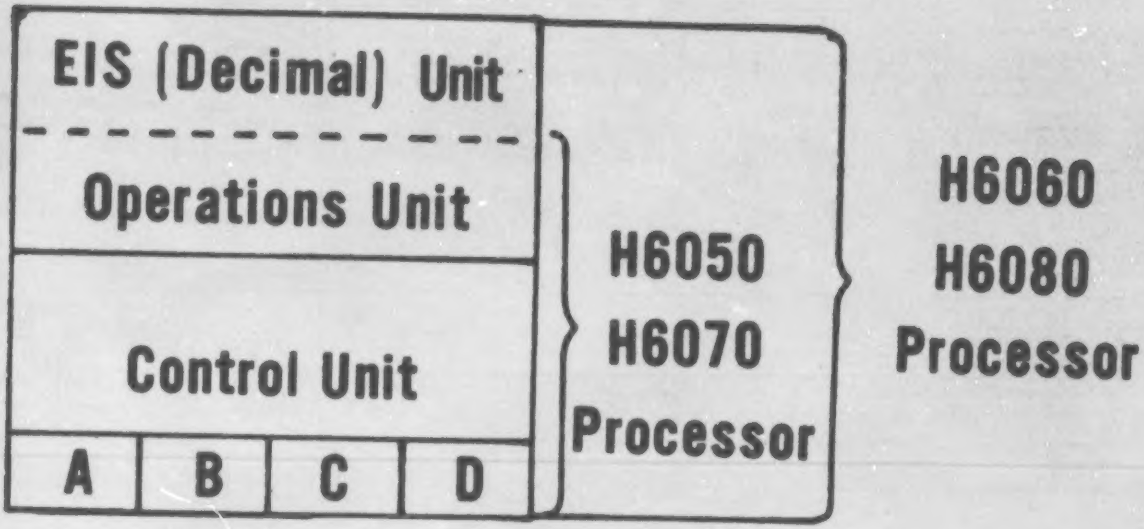
0,1
4,5
8,9

SCU

Store B

2,3
6,7
10, 11

Each store can have 128K - which means a fully configured H6000 can have a maximum of 1024K or 1,048,576 words of storage.



EIS (Decimal) Unit

Operations Unit

Control Unit

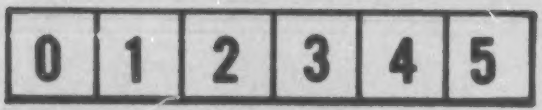
A B C D

H6050
H6070
Processor

H6060
H6080
Processor

PROCESSOR MODULE

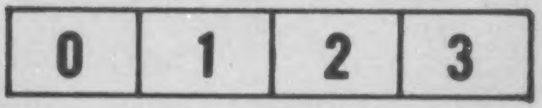
BCD



36 bits

6 BCD characters per word

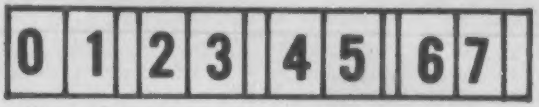
ASCII



36 bits

4 ASCII characters per word

PACKED DECIMAL

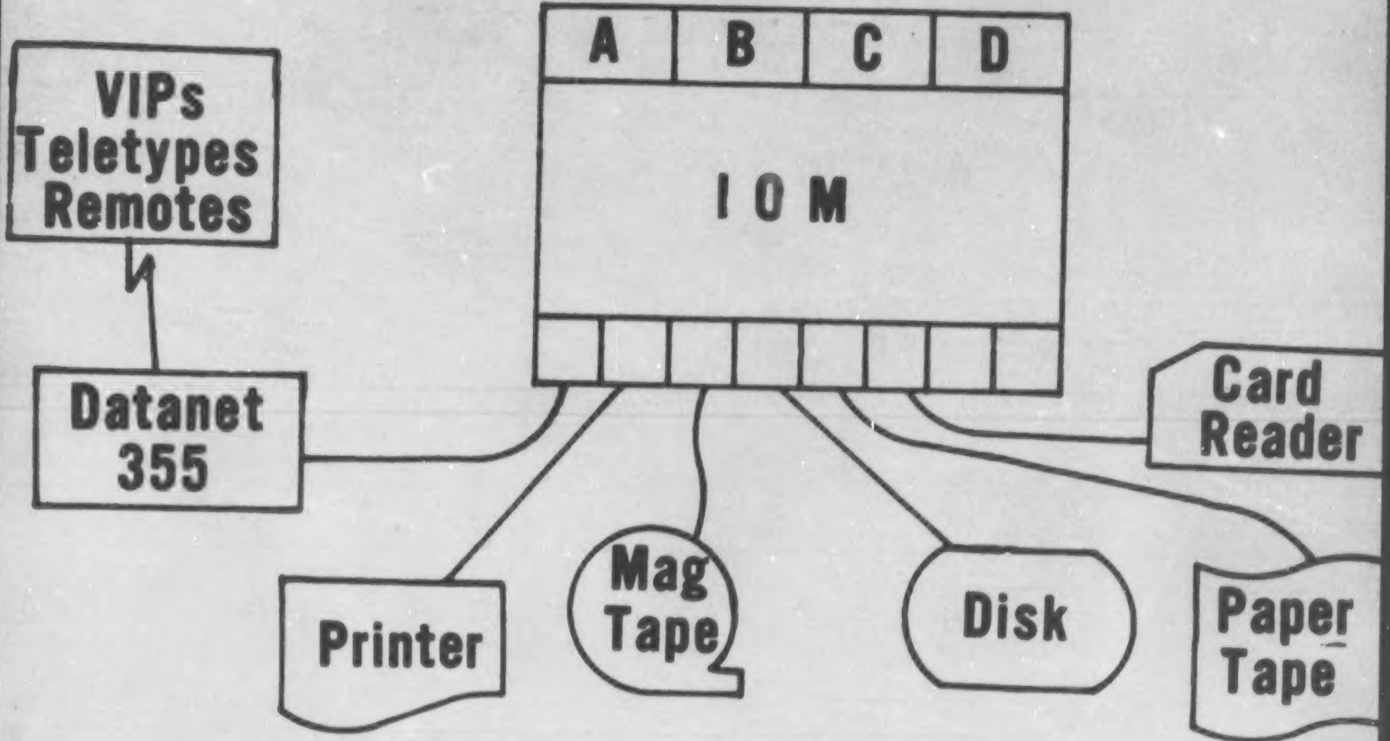


36 bits

8 characters per word

CODES FOR DATA REPRESENTATION

IOM MODULE


A
B
C
D
I O M
VIPs
Teletypes
Remotes
Datanet
355
Printer
Mag
Tape
Disk
Paper
Tape
Card
Reader

RADIX - The number of symbols that can occupy a digit (counting) position.

POSITIONAL VALUE - The power to which a symbol is placed to the left or right of the radix point.

SYMBOLIC VALUE - A value represented by a unique digit symbol of a numbering system.

EXPANDED SCIENTIFIC NOTATION - The expansion of a number showing its positional value times its symbolic value.

48

-(Most significant digit)- The left-most non-zero digit. Positive powers: farthest from the radix point. Negative powers closest to the radix point.

LSD -(Least significant digit)- The right-most non-zero digit.

RECIPROCAL POWER -Negative powers.

$(1/10^1=10^{-1}=.1 \; ; \; 1/10^2=10^{-2}=.01)$

DECIMAL TO BINARY BY EXPANSION

STARTING AT THE MSD, SUBTRACT THE BINARY POSITIONAL VALUE FROM THE DECIMAL NUMBER, ACCOUNT FOR THE SUBTRACTION WITH THE BINARY DIGIT, AND CHECK THE NEXT DIGIT TO THE RIGHT TO SEE IF AN APPROPRIATE VALUE MAY AGAIN BE SUBTRACTED. REPEAT UNTIL THE DIFFERENCE IS EQUAL TO ZERO.

BINARY TO DECIMAL BY SUMMING

SUM THE DECIMAL EQUIVALENT OF EACH NON-ZERO BINARY POSITIONAL VALUE.

50

82

DECIMAL TO BINARY CONVERSIONS

(INTEGERS) DIVIDE BY THE RADIX (2) AND SAVE REMAINDERS, LSD FIRST.

(FRACTIONS) MULTIPLY BY THE RADIX (2) AND SAVE OVERFLOW, MSD FIRST.

51

BINARY TO DECIMAL CONVERSIONS

(INTEGERS) MULTIPLY BY THE RADIX (2) AND ADD NEXT DIGIT TO THE RIGHT TO THE PRODUCT. REPEAT UNTIL ALL DIGITS ARE USED UP.

52

BINARY TO OCTAL CONVERSIONS

STARTING WITH THE RADIX POINT, GROUP THE BINARY DIGITS IN THREES AND EXPRESS AS OCTAL.

OCTAL TO BINARY CONVERSION

KEEP THE RADIX POINT AND EXPRESS EACH OCTAL DIGIT AS ITS EQUIVALENT THREE BINARY DIGITS.

COMPLEMENT --

The inversion of a numeric value derived by subtracting that numeric value from the number of counting combinations possible by the value's positional power.

For example – a 3 digit decimal number has 1000 counting combinations, so any 3 digit decimal value subtracted from 1000 will give the value's complement.

86

ADDITION AND SUBTRACTION

ADDITION

a. When adding like signs perform a straight addition and retain the sign.

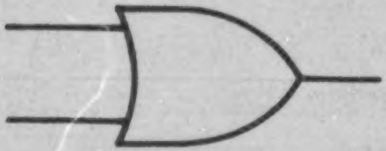
b. When adding unlike signs, subtract the smaller value from the larger and retain the sign of the larger value.

SUBTRACTION:

When subtracting like or unlike signs, change the sign of the subtrahend then proceed according to the rules of addition.

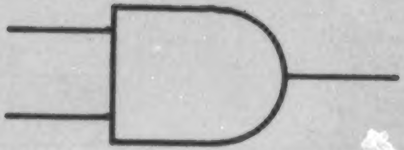
OR - +, V,

0011
0101
0111



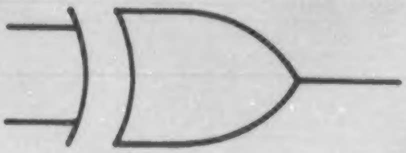
AND - ., Λ,

0011
0101
0001



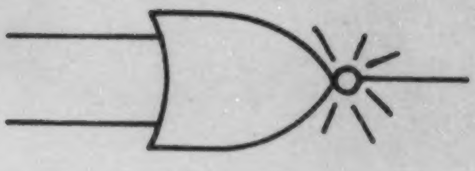
EXCLUSIVE OR -

0011
0101
0110

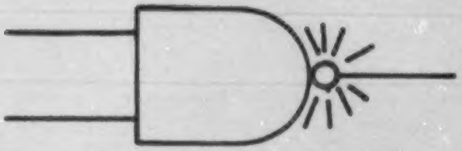


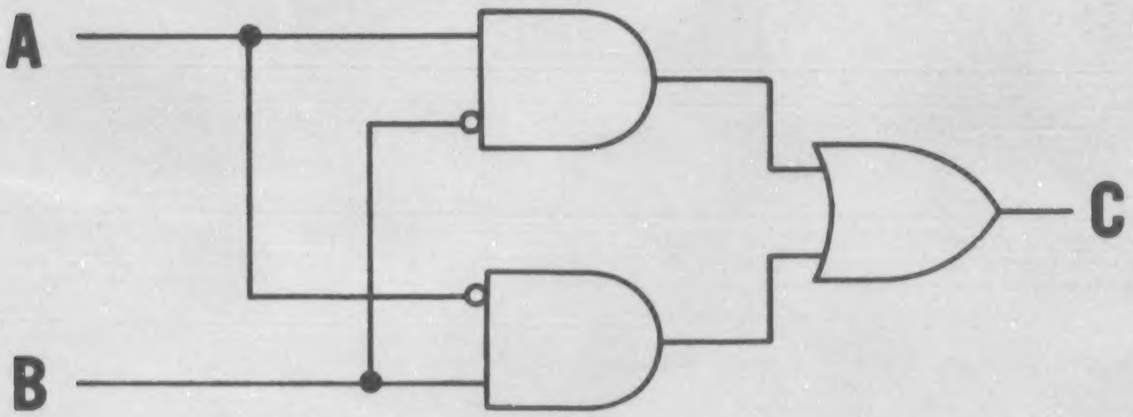
$A \bullet B' + B \bullet A' = C$

NOR - A + B = $\overline{C}$

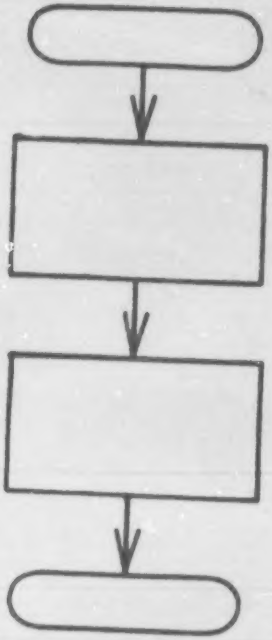


NAND - A • B = $\overline{C}$

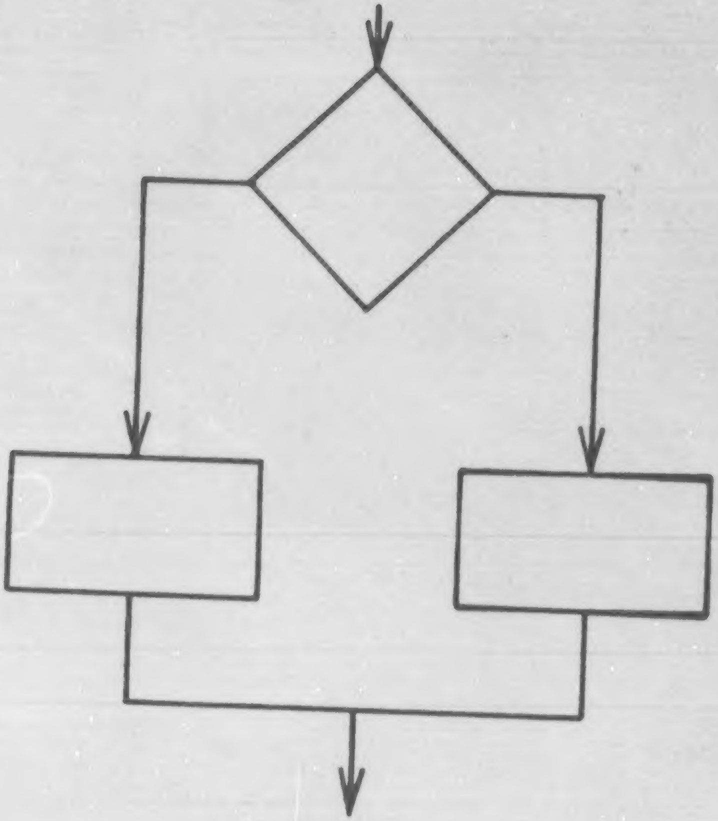




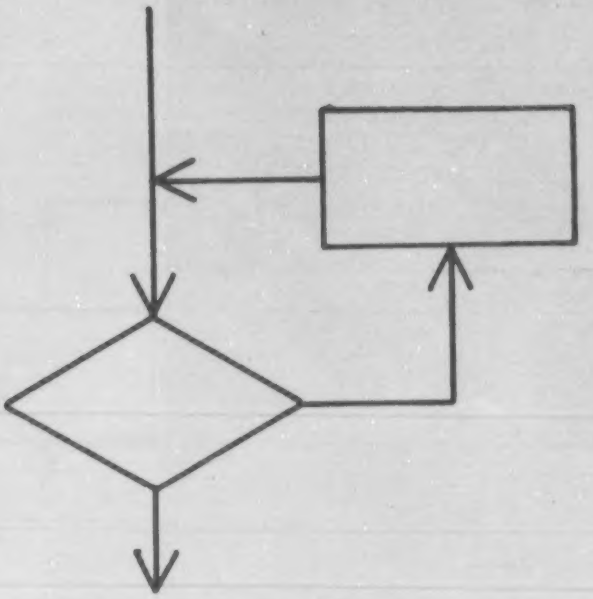
A	B	C
0	0	⇨
0	1	⇨
1	0	⇨
1	1	⇨



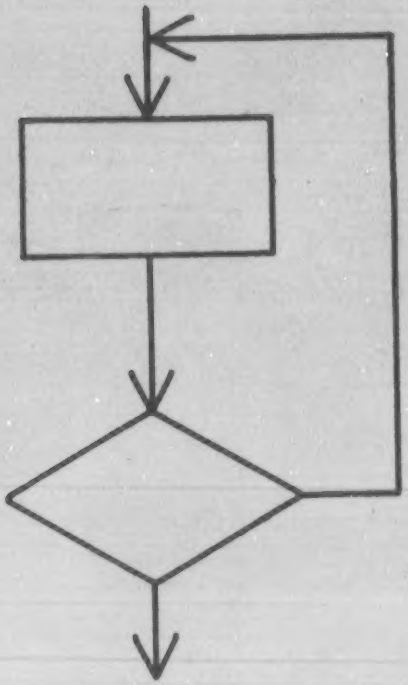
SEQUENCE



IFTHENELSE

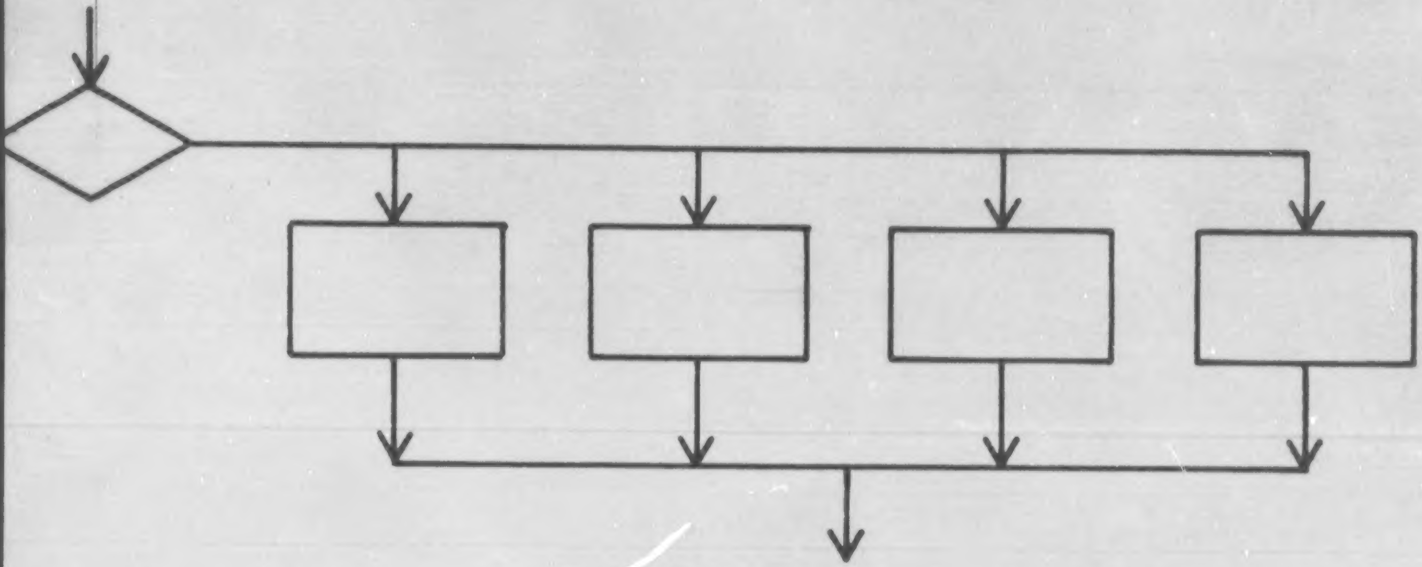


DOWHILE



DOUNTIL

92



CASE

SEQUENCE FLOWCHART

A SEQUENCE FLOWCHART IS SIMPLY A LOGICAL SEQUENCE OF OPERATIONS TO BE PERFORMED ONLY ONE TIME.

62

BRANCHING FLOWCHART

A BRANCHING FLOWCHART IS SIMPLY A SEQUENCE FLOWCHART WITH THE ADDED DIMENSION OF MAKING ONE OR MORE DECISIONS IN THE SEQUENCE OF OPERATIONS TO BE PERFORMED.

LOOP FLOWCHART

A loop flowchart is a branching flowchart with the addition of a loop. The four parts of a loop are:

1. Initialize - set the counter to zero (performed outside the loop);
2. Test - make a decision to see if the number in the counter is equal to the total number of times the operation is to be performed;
3. Perform - this includes the part of the flowchart that shows the operations to be performed; and
4. Modify - change the counter to show the number of times the operation has been performed.

96

SEARCH FLOWCHART

A SEARCH FLOWCHART IS ONE WHICH LOOKS FOR A SPECIFIC RECORD WITHIN A DATA FILE, BASED UPON SOME KEY FIELD. THE METHOD USED TO SEARCH FOR DATA WITHIN A FILE IS BASIC TO ALL OTHER MANIPULATIONS OF RECORDS WITHIN A FILE, SINCE YOU MUST LOCATE THE DESIRED RECORD BEFORE YOU CAN OPERATE ON IT.

STUDY GUIDE
E3ABR51131B 000
KDA-470

97

Technical Training

Programming Specialist (Honeywell)

COMPUTER PROGRAMMING PRINCIPLES

June 1976

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler 8-4372

DO NOT USE ON THE JOB

SG E3ABR51131B-000
KDA 470
June 1976

COMPUTER PROGRAMMING PRINCIPLES

CONTENTS

	Page
CHAPTER 1 - Introduction to Computers	1-1 through 1-26
CHAPTER 2 - Computer Mathematics	2-1 through 2-12
CHAPTER 3 - Concepts of Data Description	3-1 through 3-3
CHAPTER 4 - Problem Solving and Flowcharting	4-1 through 4-31
APPENDIX A - Flowchart Symbols for Data Processing	A-1 through A-5
APPENDIX B - Stored Program Instructions	B-1 through B-5
APPENDIX C - Segmenting Structured Programs	C-1 through C-2
APPENDIX D - Steps in Program Problem Solving	D-1 through D-3
APPENDIX E - Standard Character Set	E-1
APPENDIX F - Powers of 2	F-1 through F-2

CHAPTER 1

INTRODUCTION TO COMPUTERS

OBJECTIVES

1. Identify the major ideas of computer development in the four eras of data processing history and in the three generations of computer systems.

2. Identify the basic differences of digital, analog, and hybrid computers.

3. List and briefly explain the functions of the elements of a digital computer.

4. Identify the characteristics of machine, assembler, and compiler languages.

INTRODUCTION

The term "data processing" is relatively recent in origin; however, this does not mean that the activity itself is new. There is evidence that the need to process data originated at the beginning of recorded history when man's activities first exceeded his ability to remember. Through the years, commercial and government agencies have created the need for keeping records of some type.

Data processing refers to the recording and handling that are required to convert data into a more refined or useful form. These tasks have been referred to as recordkeeping or paperwork. They have always been accepted as routine clerical activity. With the introduction of more sophisticated electromechanical and electronic business machines in recent times, the terms "paperwork" and "recordkeeping" have been replaced by the phrase "data processing."

To fully understand and appreciate the significance of data processing as we know it today, we must examine the history of data processing.

INFORMATION

HISTORY OF DATA PROCESSING

Manual Era

The very earliest type of data processing known to man was the use of fingers, sticks, and stones as "indicating" devices. These devices actually did nothing more than serve as reminders of a particular quantity. All of the actual calculations in this case were done mentally. In 3000BC a more sophisticated "indicating" device was developed. The abacus, as it was called, consisted of rows of beads on a wire frame representing units, tens, hundreds, etc. This ancient device is still used to a certain extent today.

Mechanical (Key-Driven) Era

In 1642, Pascal, a French mathematician, invented a hand-operated, gear-driven adding machine. It registered decimal values by rotating a wheel from 1 to 9 steps with a carry lever to operate the next higher digit wheel when the first reached 10 units.

Leibnitz, a German mathematician, advanced the idea of a machine that could multiply as well as add in 1673. In practice, however, it was not very accurate.

In 1801, a Frenchman named Jacquard used a notched card to control the weaving of a fabric on a loom. Public acceptance of his device was, however, limited because of a fear of machines.

A major advancement was made in 1833 by Babbage, an English mathematician. He advanced the idea of a machine controlled by punched cards and capable of making logical decisions. He called it the difference machine. It was never built in a large number because the technology of the day lagged far behind the concepts involved.

Punched Card Accounting Machines (PCAM) Era

The 1880 census took 7-1/2 years to complete by hand. Hollerith, an American working for the Census Bureau, proposed recording, tabulating, and analyzing facts by machine. For the first time, data was punched into 3" x 5" cards according to a code invented by Hollerith, and punched card machines were designed which could handle this new medium. The 1890 census took only 2-1/2 years despite a population increase. In 1896, Dr. Hollerith formed the Tabulating Machine Company to promote his invention commercially. In 1911, his company merged with the International Time Recording Company and the Dayton Scale Company to form the Computing-Tabulating-Recording Company. In 1924, the name was changed to the International Business Machines Corporation (IBM).

Through the succeeding years, IBM became the major manufacturer of PCAM machines. There are six classifications of PCAM machines used today.

1. Keypunch - Used to punch the Hollerith code into cards.
2. Reproducer - Used to duplicate decks of punched cards.
3. Interpreter - Prints on cards, the information punched into those cards.
4. Sorter - Resequences punched decks.
5. Collator - Merges punched decks together.
6. Accounting Machines - Produce printed listings from decks of cards.

Although PCAM machines were a vast improvement over earlier methods and devices, they still had several major disadvantages. The equipment was mechanically too slow. Processing was done in stages from one machine to another. Processing on PCAM machines also required close human supervision.

Electronic Era

In 1944, Aiken built a machine at Harvard University. It was called the Automatic Sequence Controlled Calculator (Mark I) and could perform long series of arithmetic and logical problems. It was designed for use by engineers, physicists, and mathematicians. This machine was an outgrowth of Babbage's ideas and blueprints. It was not strictly electronic because it made decimal calculations through electromechanical means (relays and switches).

In 1945, Mauchly and Eckert built the first truly electronic computer, with no moving parts. It was called the "ENIAC" (Electronic Numerical Integrator and Calculator) and contained 18,000 vacuum tubes. It was developed to perform great quantities of statistical calculations for weather data. The ENIAC performed 5,000 ten-decimal-digit calculations per second and had a multiplication speed of up to 300 calculations per second. The addition of two numbers which took 300 milliseconds on Mark I could be done in .2 milliseconds on ENIAC.

The next major development in the electronic era came in 1952. Eckert and Mauchly developed EDVAC (Electronic Discrete Variable Automatic Computer). It consisted of only 3500 vacuum tubes, and was the prototype of serial computers. One of the major advancements of the EDVAC was that operations were performed entirely by a stored program rather than prewiring. With the development of EDVAC it was realized that computers could be useful in areas other than those of a scientific nature.

In 1946, Mauchly and Eckert had organized their own company, the Electronic Control Company. In 1951, they produced the first truly commercial computer. The UNIVAC I (Universal Automatic Computer) was installed in 1951 at the U.S. Bureau of Census. The first business use of computers was in 1954 when UNIVAC I was installed at GE Appliance Park, Louisville, Kentucky. The Electronic Control Company was later acquired by the Remington Rand Corporation, which became the UNIVAC Division of Sperry Rand Corporation.

The second commercial computer was the CRC 102 that was produced in 1952 by the Computer Research Corporation. Closely following this computer in 1952 was the IBM 701.

The first computer developed for both scientific and business applications was the IBM 650, which was first used in 1954 in Boston, Massachusetts.

COMPUTER GENERATIONS

In the discussion of the history of data processing, we will discuss the first, second, and third generation computers. The following text explains these areas of computer development and describes certain possible characteristics of fourth generation systems.

First Generation (1946 - 1958)

First generation computers are characterized by the use of vacuum tubes, which made them much faster than electromechanical data processing machines. Most first generation computers used internally stored programs, adding to their flexibility and reducing setup time, although virtually all programming was done in machine language. Main memory began its advancement from vacuum tubes to magnetic drum and magnetic core.

The computers were huge, and required strict air-conditioning standards because of the heat produced by the vacuum tubes. Input was restricted primarily to punched card and paper tape, although magnetic tape was beginning to be used.

Second Generation (1958 - 1965)

The second generation marked the advent of large-scale computers. These computers had large memories and microsecond access time.

The main reason for these improvements was the replacement of the vacuum tube by the transistor (1958). This resulted in reduced physical size of the computer, improved speed and reliability, lowered cost, less power, and less air conditioning required.

Storage was usually magnetic core, although some small systems still used magnetic drum. Thin film was introduced in 1960.

A greater range of peripherals was introduced, including display devices, faster printers, and data transmission devices for long-range communication between the computer and remote terminals. The latter made on-line data processing possible.

Third Generation (1965 - 1970)

It became increasingly evident that further increases in speed were dependent upon the speed at which electricity can travel. To cope with this problem and to maintain a high degree of reliability, integrated circuits were developed. They are considered the main characteristic of third generation computers. These circuits were implemented in late 1964 with the announcement of whole new families of computers. These included the IBM System 360, RCA Spectra 70, Burroughs B3500, and others. With this decreased circuit size, it was possible to operate in the nanosecond range (one nanosecond equals 10^{-9} second).

The integrated circuit concept involves the synthesizing of electrical components such as transistors, resistors, and capacitors, into a single tiny block to form a complete circuit function. An example is shown in figure 1-1. The left-hand portion illustrates 15 transistors and 13 resistors of the type replaced by the integrated circuit on the right.

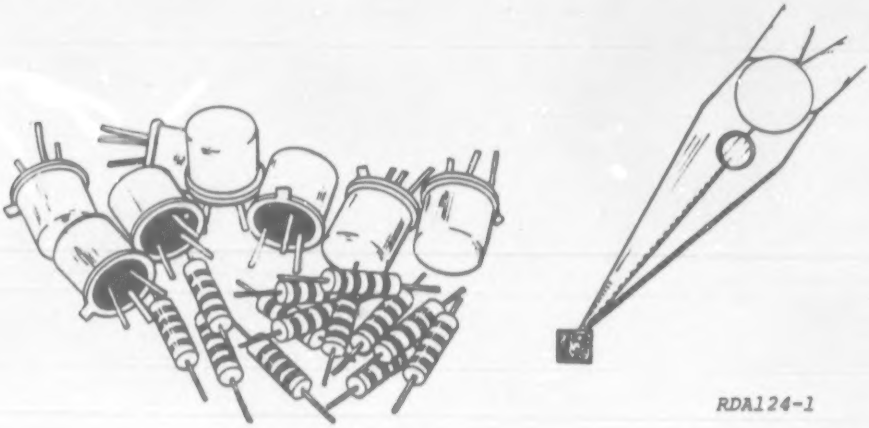


Figure 1-1

Greater emphasis was placed on hardware modularity, operating systems, and increased input-output capabilities, including optical scanning and Magnetic Ink Character Recognition (MICR). This has produced computer systems which possess increased capabilities over previous generations. One significant advantage is the ability of third generation computers to process more than one job simultaneously. Increased speed and the use of sophisticated software made time-sharing terminals, multiprogramming, multiprocessing, and real-time processing practical. The third generation computers are more flexible than earlier systems, suited to both scientific and business data processing.

Future Generations

While definite characteristics of fourth generation computers are not yet known, certain tendencies are known. These tendencies are:

1. Faster - Most engineers are predicting a five-fold increase in circuit speeds over the next five years. There seems to be no question that significant increases in central processor speeds will continue to occur for many years to come.

2. Less Expensive - In 1960 a transistor (the size of a pencil eraser) cost an average of $1 each. In 1970 large-scale integrated circuits were used, containing more than 200 transistors, costing no more than one cent each. By the early 1980s the use of MOS technology (metal-oxide-semiconductor circuits containing a million transistors) is expected to reduce the cost of components to .003 cent.

3. More Reliable - Since the production costs of circuits will be significantly reduced and because standardization in circuit design will be increased, systems will have the ability to choose alternate circuits if failure occurs. This ability to continue functioning, even though one or more failures have occurred in the hardware, is called "self-healing."

4. Better Input-Output Devices - Speed and reliability of input-output devices will increase. These devices will be located near the source of data and will reflect a trend toward media that can be understood by both man and machine. Examples of this type of media are printer devices that will accept typed data and can also respond on the same device with typed information, visual display devices such as CRT (cathode-ray tube) and standard TV sets, audio devices, and OCR (Optical Character Recognition) devices that can read printed data of a variety of character fonts.

TYPES OF COMPUTERS

One method of classification divides computers into three general types--analog, digital, and hybrid--and into two categories of application--special-purpose and general-purpose.

The Analog Computer

The analog data processing machine, as the name implies, processes work electronically by analogy. It takes measured amounts of information, performs a set routine of processing, and presents this information in measurable form.

An example is a speedometer on the dash of a car. The rotations of the car wheels are transferred through a flexible cable to the inertial-governor device under the dashboard. This device then interprets the rate of speed of the car in terms of a dial reading.

The analog computer can perform only that function for which it was designed, and it is not as accurate as the digital computer. It is often compared to a slide rule which is only as accurate as the exactness of the measurement of its scale. It is ideally suited for certain engineering applications, but unsuitable for calculations where accuracy is required to the exact digit as in payroll calculations.

Digital Computers

The digital computer accepts information in the form of coded alphabetic and numeric characters. It then processes this information in accordance with a predetermined instruction sequence that can be varied as required. We are dealing with a discrete variable; one that represents exactly what it stands for. In an analog machine we are dealing with an indiscrete variable; one that represents an approximation of a quantity.

Digital computers can be split into groups according to the internal memory system of the computer and its coding structure. The divisions are based on the type of coding system prevalent in the machines' operations. Two major coding systems are binary and decimal. Other coding systems exist, but they are merely variations of these two.

Hybrid Computers

Some computers are designed with both analog and digital capabilities. These systems are hybrids. An example of this type of computer system is the SAGE system which is an early warning system against surprise air attacks. Process control systems such as the computer systems used in controlling the refining of petroleum products are also examples of hybrid computer systems.

Special-Purpose Computers

Another widely used method of classifying machines is to categorize them as special-purpose and general-purpose machines. Special-purpose computers are designed for a specific operation. The SAGE computer illustrates this scheme of classification. The SAGE system was specially designed and built as an early warning network against enemy attacks. In the Air Force, this type of machine is usually limited to applications in the weapons system area.

General-Purpose Computers

General-purpose machines are not specifically designed for any applications and, hence, adapt to a variety of jobs. Although they are adaptable, certain machines are better suited to do scientific applications and others to business-logistical applications.

The scientific computer takes in small quantities of input, which may be drawn from business files or from facts about current operations. It performs vast amounts of mathematical and logical operations and provides a small amount of output. The most important consideration is a high-speed processing capability. The IBM 7090, used in tracking earth satellites, and Remington Rand's LARC, used at the Los Alamos Research Center for calculation of trajectories, are examples of scientific computers.

The business-logistics computer takes in vast quantities of data, performs a relatively small amount of processing, and puts out large quantities of output. File contents are analyzed to find relationships and answer simple inquiries. It is essentially a scientific computer with the additional capability for large volume data input and output at a relatively high speed. In this type of machine, input and output speeds are more important than processing speed.

When determining the type of hardware to be used, the method of processing desired must also be considered. Many varieties of methods for processing data are available today. The major consideration in deciding which method to use is the economic factor. The more powerful the processing desired, the more money that must be spent.

DIGITAL COMPUTER ELEMENTS

For a digital computer to operate, certain elements are required for the proper handling and manipulation of data, just as a man needs certain tools to perform arithmetic tasks. This comparison is easily supported by describing the elements of the computer that correspond to a man working at a desk. Let's assume that the man is a clerk working in a payroll office and is computing the net pay of various individuals. The IN box on his desk contains the pay rates of the personnel involved plus miscellaneous data such as the initiation of bond deductions, etc. A digital computer has an INPUT ELEMENT which is capable of accepting various types of data and presenting it to the computing portion of the equipment. The clerk has several tables to which he refers such as tax deduction tables, standard weekly deductions, etc. In addition, he has a pad and paper on which he notes the deductions applicable to each employee. In a digital computer, the MEMORY ELEMENT would serve as the temporary storage device for all these facts. The actual computation of an individual's salary is done with a desk calculator. This function is the same as that performed by the ARITHMETIC ELEMENT of a digital computer. Once the net pay of each person has been calculated, the clerk fills out a standard form for each employee which contains the employee's name and the amount due. He then places all their forms in the OUT box on his desk, thus completing his job. The OUTPUT ELEMENT of a digital computer accepts the results of computation by the arithmetic element and presents the results in a form recognizable by the user. Of course, all the actions of the payroll clerk are controlled and coordinated by his nervous system. The CONTROL ELEMENT coordinates the actions of a digital computer and is connected to all the other elements. From this discussion, we see that a digital computer is essentially composed of the following elements:

1. Input.
2. Memory.
3. Arithmetic.
4. Output.
5. Control.

The operational elements listed above are the elements required by typical digital computers. The following paragraphs describe these elements in more detail and explain the tasks performed by each.

Input Element

The input element is capable of accepting data in a variety of forms and converting it to a standard format which the other computer elements can use. For example, in figure 1-2, a typewriter input device might be used; accordingly, the operator would type out the data and instructions. The typewriter would have switches connected to each key which would convert the hitting of a key into an electrical impulse. The electrical impulse might then be converted to a binary code so that the computer could work with it. Other common types of input devices are punched card readers, magnetic tape readers, paper tape readers, and such automatic input units as telephone lines which transmit data from remote locations. The input element provides one-way communication between the external input devices and the other computing elements. Data and instructions are fed to a computer through an input element.

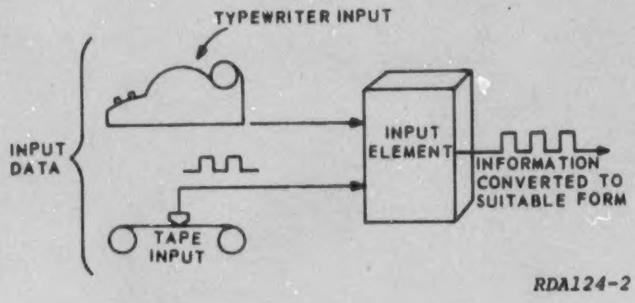


Figure 1-2. Input Element - Receives Information and Converts it into Usable Form

Memory Element

As explained previously, operations in a digital computer are carried out in step-by-step fashion. For this reason, some of the information fed into a computer must be stored prior to actual usage. The facilities required for storing information in a computer are included in the memory (sometimes called main storage) element.

Information fed into a computer includes:

1. Particular items of data to be processed.

2. Instructions (known as the program) for performing the particular data processing operations required.

3. Reference data.

The memory element comprises a large number of storage locations in which information can be stored until it is needed by one of the other elements. Each of these locations has an absolute address assigned so that it may be selected by the computer for insertion or extraction of data. For instance, a typical computer instruction might be to "add the quantity which is stored in location 1000." The instruction which stated that address 1000 contained the desired OPERAND is also stored in the memory element.

Many types of storage devices such as magnetic cores, magnetic tapes, magnetic drums, acoustic delay lines, and cathode-ray tubes are used in memory elements. At present, magnetic cores are the most popular device, primarily because of their high speed and stability.

Arithmetic Element

Since the purpose of a digital computer requires that the machine perform arithmetic operations on the input data, obviously a digital computer must contain an element that can accomplish these operations. This is the arithmetic element. All data to be operated on arithmetically must enter this part of the computer. Likewise, most instructions determining what computations are to be performed must control the arithmetic element (see figure 1-3).

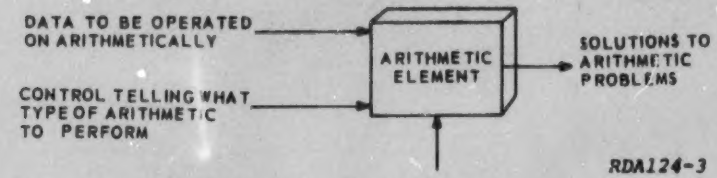

DATA TO BE OPERATED ON ARITHMETICALLY
CONTROL TELLING WHAT TYPE OF ARITHMETIC TO PERFORM
ARITHMETIC ELEMENT
SOLUTIONS TO ARITHMETIC PROBLEMS
RDA124-3

Figure 1-3. Arithmetic Element - Data Enters and is Processed by this Element

100

Theoretically, it would be possible to build an arithmetic element which could perform most mathematical operations directly, just as a man performs them. This, however, would require a very large and complicated arithmetic device; consequently, it is never done. Instead, the arithmetic element is usually designed to perform only a few fundamental operations such as addition, subtraction, multiplication, and division. It can be made to perform almost any complex mathematical operation by simply breaking the operation down into these fundamental steps.

Output Element

The results of a digital computer's operations must be delivered to the user of the machine in an appropriate form. The element that accomplishes this transfer is the output element. The results of a computer's operations, however, are not necessarily in the form best suited for use outside the machine. Hence, an output element may include facilities for converting the results of the computer's operations into the form of output data best suited to the user of the machine. For example, in figure 1-4, the answer to the problem might enter the output element in the form of binary electrical pulses. The output element may then convert these pulses to voltages that operate either an electrically operated typewriter or a printing machine to print the final answer.

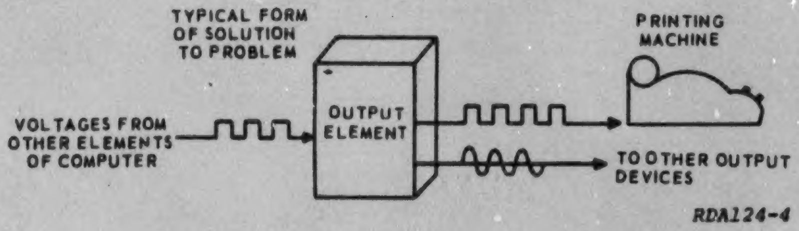

TYPICAL FORM OF SOLUTION TO PROBLEM
PRINTING MACHINE
VOLTAGES FROM OTHER ELEMENTS OF COMPUTER
OUTPUT ELEMENT
TO OTHER OUTPUT DEVICES
RDA124-4

Figure 1-4. Output Element - Converts Computer's Answers into Form Usable by External Output Devices

Common types of output devices are line printers, card punches, magnetic tapes, and visual indicators.

The output element provides a one-way communication between the other elements of the computer and the external output devices.

Control Element

The entire sequence of operations by the computer is predetermined by the program and the construction of the computer. The program, coded in the digital language, is inserted through the input element and stored at specific addresses in the memory element. The element for interpreting and carrying out instructions contained in the program is the control element.

There must be a definite sequence for the flow of data during processing by a digital computer. All of these operations are done at the command of the control element. For example:

1. Data must be inserted into particular storage locations and then used in correct sequence at appropriate times.

2. The arithmetic element must also be "told" what operations to perform on the data and in what order to perform them.

3. The results of the arithmetic operations must be routed to the appropriate storage or output locations.

4. The transfer of all output data to the output element must be properly controlled to insure the required sequence of information.

Figure 1-5 is the Honeywell 6000 series configuration of digital computer elements. Notice that the processor and IOM access memory separately. This is a major third generation concept, called asynchronous I/O.

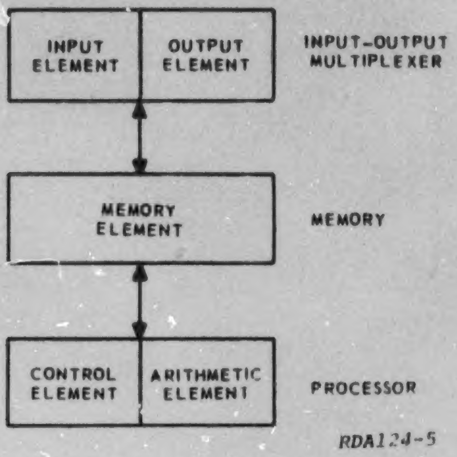


Figure 1-5. Honeywell 6000 Series Configuration of Digital Computer Elements

PROGRAM FLOW IN COMPUTER SYSTEMS

The various elements of a digital computer have just been covered. But what actually happens to a program from input to a computer to output? This section will answer that question after a short discussion of a simple system and related terms. Figure 1-6 is a block diagram of a digital computer.

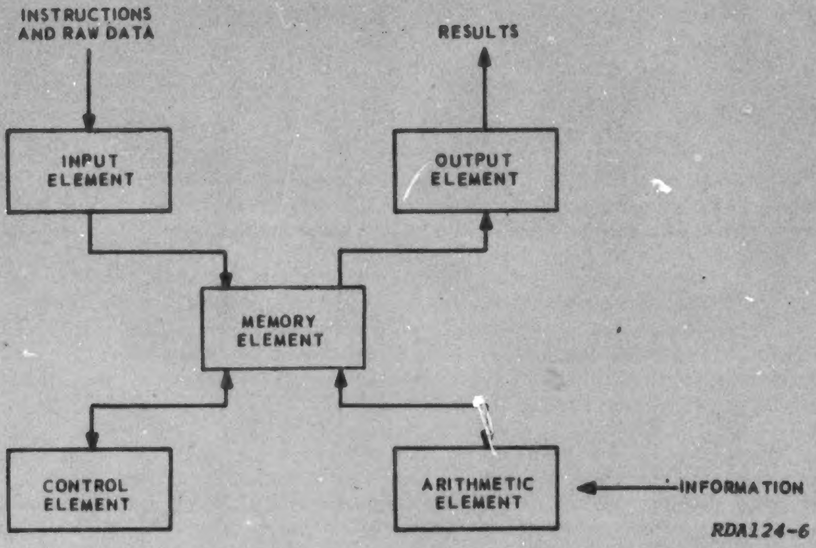


Figure 1-6. Digital Computer, Block Diagram

Basically, any computer may be thought of as a simple system (see figure 1-7). Data is input into this system, manipulated by a process, and output as information. A close look at the last sentence will reveal a subtle difference between data and information. Data is more or less a set of facts about something, which have little meaning by themselves. Information, on the other hand, is data which has been transformed by some process into a meaningful form. In short, information is data which means something, not just a lot of facts. The simple system in figure 1-7 is controlled by a programmer (to use a computer-related term).

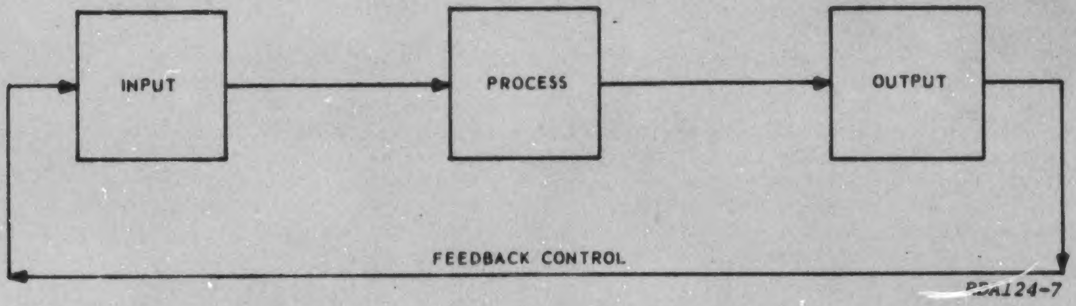


Figure 1-7. A Simple System

The point of this discussion is that any system, whether it be simple or complex, is only as good as the individual that controls and directs its operation--the programmer. Translating this to computers--a computer is really just a nonthinking machine. It will do exactly what it is instructed to do and nothing more. The programmer must tell this "idiot" what to do by means of a program. Note that a program is the complete sequence of coded instructions necessary to process data into information.

Now back to the question on what happens to a program in the computer system. To answer this question, trace the flow of a program through a digital computer (see figure 1-6). This example is based on the assumption that the program will be a deck of cards and will yield a printer output.

The program is first read by the card reader and sent to the input element. The input element converts the characters/symbols from program card deck into a machine usable code (i.e., BCD, ASCII, etc.) and sends this coded program to the memory element. The memory element accepts this code at specific intervals and stores it at address locations within memory.

Note that two paths lead out of the memory element--one to the arithmetic element and one to the control element. This is because a program consists of both instructions and data (to be operated on by instructions). Therefore, the program instructions go to the control element and the program data goes to the arithmetic element. Program execution is, simply put, a process of performing each program instruction by the control element. This execution involves calculating quantities in the arithmetic element and storing quantities of data in the memory element.

Upon completion of the internal processing of the program, the resulting information is sent to the output element. The output element then reconverts the information from machine code to characters/symbols for output by the printer.

This completes the flow of a program through a digital computer as seen in figure 1-7. Although the example was oversimplified, it does give some idea of what happens to your program in the machine.

PROGRAMMING LANGUAGES

Before the physical components of an electronic data processing system can solve any problem, the programmer must be able to communicate the instructions of his program to the hardware. Both the data and the instructions are stored in the computer in configurations of 1 bits and 0 bits. The characters these bits represent make up the language of the computer (called machine language or absolute language). Though programmers can, and sometimes do, communicate with the computer in machine language, the process of writing computer instructions in binary representation is cumbersome and time-consuming. For instance, if you wanted a computer to add something, you might have to write an instruction like this: 10110101110100001.

Because of the abstract nature of machine language and the great difficulty in writing programs in this language, program languages, structured in either a near-English format or mathematical format, have been developed. Programming languages constitute part of the various programming aids (called software) that a manufacturer provides with the computer's hardware.

Machine Language

Machine language is the most elementary form of coding. It is the lowest level language in which we can program. It is the only language a computer can "understand" without any translation. Each machine language instruction has an equivalent hardware circuit to perform the specified operation.

Machine language programming is the most difficult for the programmer, simply because he must give the computer instructions it can understand without further translation.

Long programs written in machine language involve a tedious amount of clerical work. The programmer must develop a list of addresses and their associated data variable names, keep track of storage allocations for data, define fields and records properly, prevent the program instruction from overlapping data, worry about a loader program to get his main program into memory and executing, and many other things. Early computers were programmed in this way because there was no alternative.

A programmer writing in the basic language of the computer can stay close enough to the elemental operating procedures of the computer to take advantage of some specialized techniques, but he has to contend with several major disadvantages:

1. All operation codes and operand addresses must be written in some numeric code.

2. All addresses in this code must be absolutely defined.

3. Since mere coding does not solve the original problem, the programmer is also faced with the task of transferring the coding into machine-readable form, proofreading the transformation, and loading the code into the memory of the computer.

Figure 1-8 on the following page shows a small portion of the machine language coding for a problem being run on the Honeywell 6060 computer.

Because of the knowledge and skill required to code even simple problems in machine language, programming languages were developed that were easier for a human to work with.

To bridge the gap between the machine language and the program language, a special type of conversion program is used. It tells the computer how to change the operations from the program language (the source program) to actual machine language instructions (the object program).

Several types of these conversion programs are available. The programming language that is used for a problem depends upon the particular problem to be solved and upon the hardware available. We will discuss two types of conversion programs--assemblers and compilers.

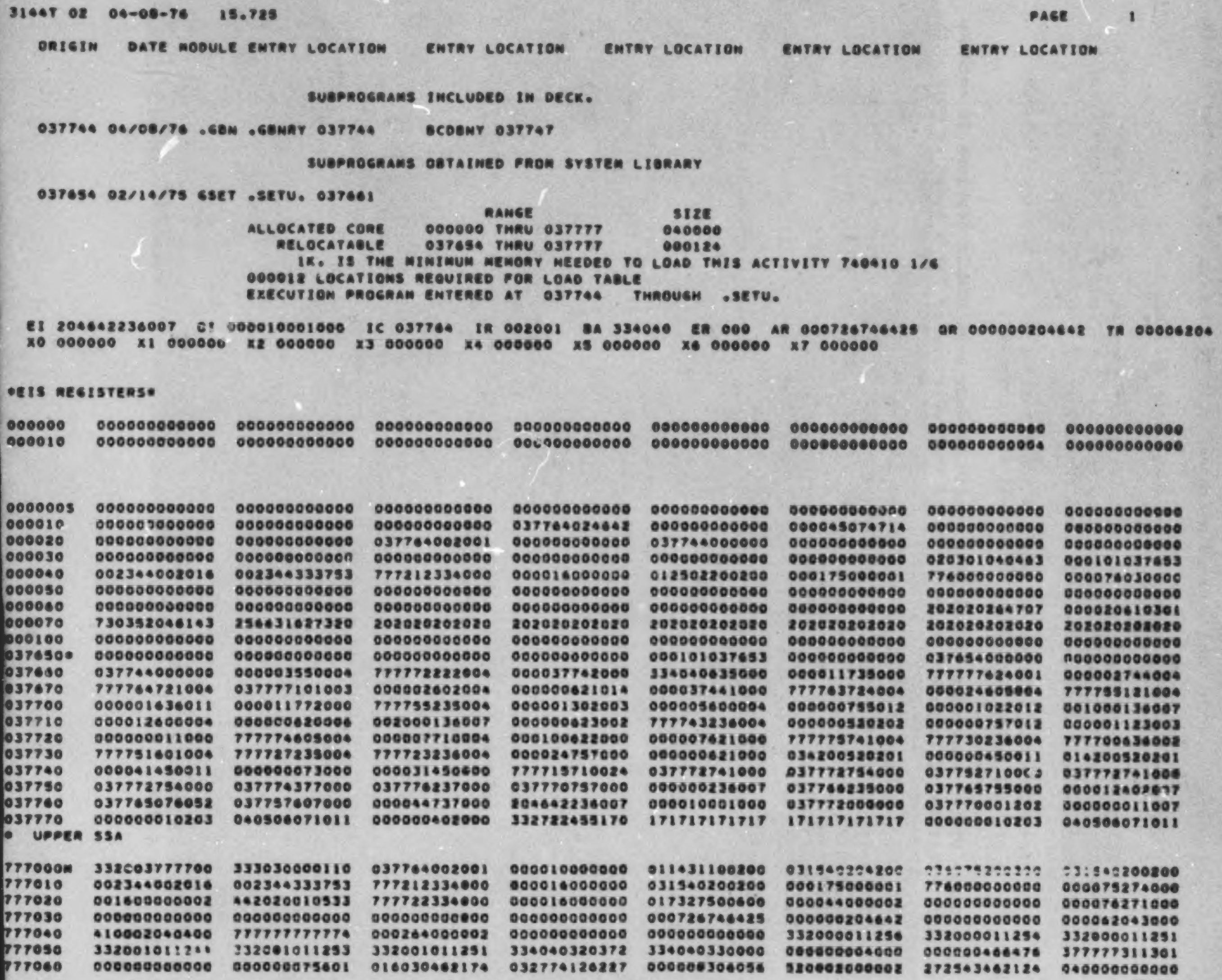
ASSEMBLY PROGRAM. An assembly program (or an assembler) converts a program written in symbolic form into a machine language program. An assembler translates item for item, i.e., it produces an object program with the same number of instructions and constants as the source program.

An assembler-level programming language generally has almost the same flexibility for internal data manipulation as machine language itself. However, an assembler-level program is usually long and tedious to write, and is machine-dependent. That is, an assembler-level program written for the IBM 360/65 would need to be completely rewritten before it could be run on the B3500 or Honeywell 6060.

COMPILER PROGRAM. A compiling program (called a compiler) converts a program written in a program language that consists of relative or symbolic coding into a machine language program. A compiler differs from an assembler in that the compiler may generate a series of machine language instructions from one instruction written in compiler program language. Thus, a compiler is a machine language program that can expand and translate the original program instructions from the compiler language format into the machine language format.

Compiler Language

A compiler language, like an assembly language, allows the programmer to write in recognizable codes. Unlike in the assembly language, the programmer does not have



```
3144T 02  04-08-76   15.725                                                                                          PAGE       1

   ORIGIN   DATE MODULE ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION    ENTRY LOCATION


                             SUBPROGRAMS INCLUDED IN DECK.

   037744 04/08/76 .GBN .GBNRY 037744     BCDBNY 037747

                             SUBPROGRAMS OBTAINED FROM SYSTEM LIBRARY

   037654 02/14/75 GSET .SETU. 037661
                                                  RANGE               SIZE
                        ALLOCATED CORE    000000 THRU 037777       040000
                           RELOCATABLE    037654 THRU 037777       000124
                             1K. IS THE MINIMUM MEMORY NEEDED TO LOAD THIS ACTIVITY 740410 1/6
                        000012 LOCATIONS REQUIRED FOR LOAD TABLE
                        EXECUTION PROGRAM ENTERED AT  037744   THROUGH  .SETU.

  EI 204642236007  C! 000010001000  IC 037764  IR 002001  BA 334040  ER 000  AR 000726746425  QR 000000204642  TR 00006204
  X0 000000  X1 000000  X2 000000  X3 000000  X4 000000  X5 000000  X6 000000  X7 000000


*EIS REGISTERS*

000000    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000010    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000004  000000000000



0000005   000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000010    000000000000  000000000000  000000000000  037764024642  000000000000  000045074714  000000000000  000000000000
000020    000000000000  000000000000  037764002001  000000000000  037744000000  000000000000  000000000000  000000000000
000030    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  020301040463  000101037653
000040    002344002016  002344333753  777212334000  000016000000  012502200200  000175000001  776000000000  000076030000
000050    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
000060    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  202020264707  000020610301
000070    730352046143  256431627320  202020202020  202020202020  202020202020  202020202020  202020202020  202020202020
000100    000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000  000000000000
037650*   000000000000  000000000000  000000000000  000000000000  000101037653  000000000000  037654000000  000000000000
037660    037744000000  000003550004  777772222004  000037742000  334040635000  000011735000  777777624001  000002744004
037670    777764721004  037777101003  000002602004  000000621014  000037441000  777763724004  000024605004  777755121004
037700    000001636011  000011772000  777755235004  000001302003  000005600004  000000755012  000001022012  001000136007
037710    000012600004  000000620006  002000136007  000000623002  777743236004  000000520202  000000757012  000001123003
037720    000000011000  777774605004  000007710004  000100622000  000007621000  777775741004  777730236004  777700636002
037730    777751601004  777727235004  777723236004  000024757000  000000621000  034200520201  000000450011  014200520201
037740    000041450011  000000073000  000031450600  777715710024  037772741000  037772754000  037752710003  037772741006
037750    037772754000  037774377000  037776237000  037770757000  000000236007  037766235000  037765755000  000012602637
037760    037765076052  037757607000  000044737000  204642236007  000010001000  037772000000  037770001202  000000011007
037770    000000010203  040506071011  000000402000  332722455170  171717171717  171717171717  000000010203  040506071011
*  UPPER SSA

777000M   332003777700  333030000110  037764002001  000010000000  011431100200  031540224200  031775200220  031540200200
777010    002344002016  002344333753  777212334000  000016000000  031540200200  000175000001  776000000000  000075274000
777020    001600000002  442020010533  777722334000  000016000000  017327500600  000044000002  000000000000  000076271000
777030    000000000000  000000000000  000000000000  000000000000  000726746425  000000204642  000000000000  000062043000
777040    410002040400  777777777774  000264000002  000000000000  000000000000  332000011256  332000011254  332000011251
777050    3320010112[illegible]  332001011253  332001011251  334040320372  334040330000  000000004000  000000466476  377777311301
777060    000000000000  000000075601  016030462174  032774126227  000000304056  520002000002  272543462124  040000000000
```

9DA124-8

Figure 1-8. Machine Language Code

every command found in machine language available to him in the compiler language. However, he does have a language which generates a whole set of machine language instructions for each compiler command. In addition, most compiler languages use familiar English language words and arithmetic symbols to generate instructions. Most importantly, since most compiler languages have been standardized, programs can be run on many different computers with little or no modification.

The first time an assembler-level or compiler-level program is run on a computer, the program passes through the computer twice; first with the assembler (or compiler) to produce the object program, then the object program version is used to process the data (see figure 1-9). On most systems the object program can be stored on disk or tape so that it need only be compiled once.

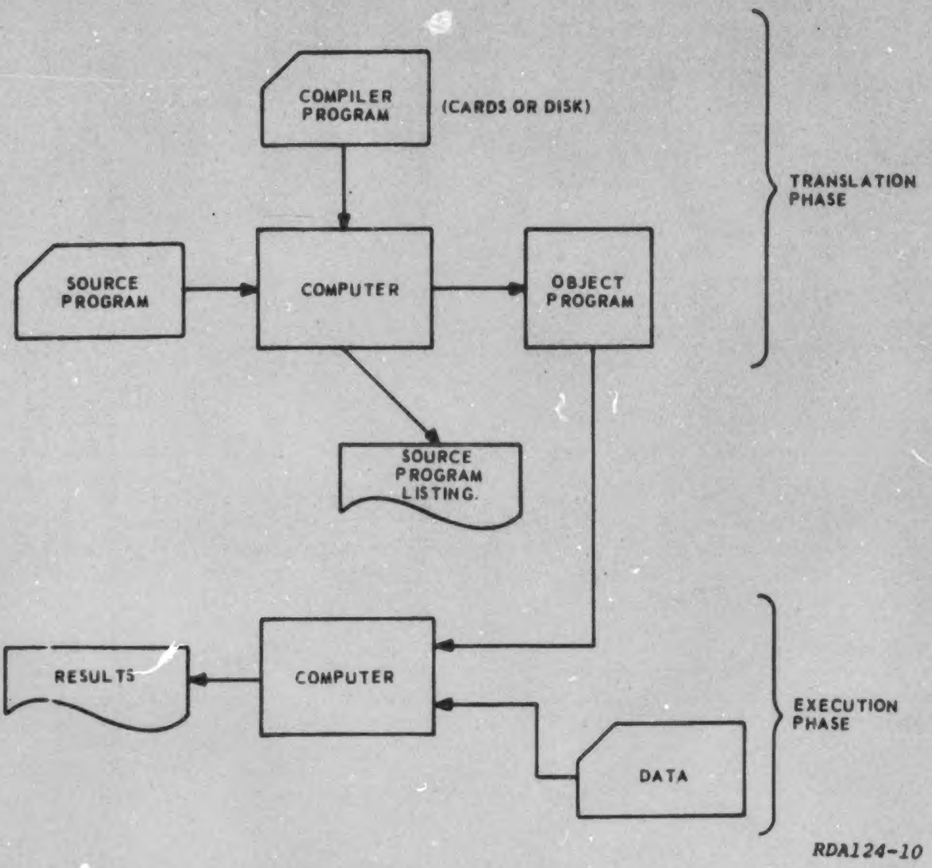


Figure 1-9

Assembler Language

In machine language, each machine code has a corresponding hardware circuit. Machine code, therefore, is the lowest level language which can be programmed.

All computers have some form of machine code, and nearly all machines have some form of assembler language. Assembler language is an extension of machine language in most cases.

Assembler language (sometimes called symbolic language) uses instructions that are written in a nonmachine language, with both operators and operands expressed as symbols. They are usually mnemonic symbols; that is, symbols that represent an original word and are easy to remember. In assembler language for the IBM 1620, for instance, "A" is the mnemonic code for addition and "MPY" is a mnemonic code for multiplication. These symbolic operators represent computer operations and machine language codes. "A" stands for machine language code 21 and "S" for code 22; "RN" symbolizes the "read numeric information" in place of code 36, and "H" is a symbolic code for "halt." The operands are symbolic names for data variables, and stand for the actual memory addresses of their values.

If a programmer wanted to add the contents of memory location 12070 to location 12060 (on the IBM 1620), in machine language the instruction would be:

```
21 12070 12060
```

while in assembler coding the programmer could write:

```
A I N
```

where I and N had been defined as representing the storage location.

During the translation phase of an assembler program (see figure 1-9), the assembler instructions are printed, the source program listing to provide documentation. On many computers, this listing will also contain the corresponding machine language instructions as a debugging aid for the programmer. Figure 1-10 shows an assembler program as written for the Honeywell 6060 computer.

The programmer is now once removed from the computer. He writes his program in a sequence of instructions that are symbolic with respect to his problem. He turns over the translation into machine language to the assembler, including the exact assignments of storage. He exempts himself from a large part of the tedious bookkeeping, spending his time on more valuable activities.

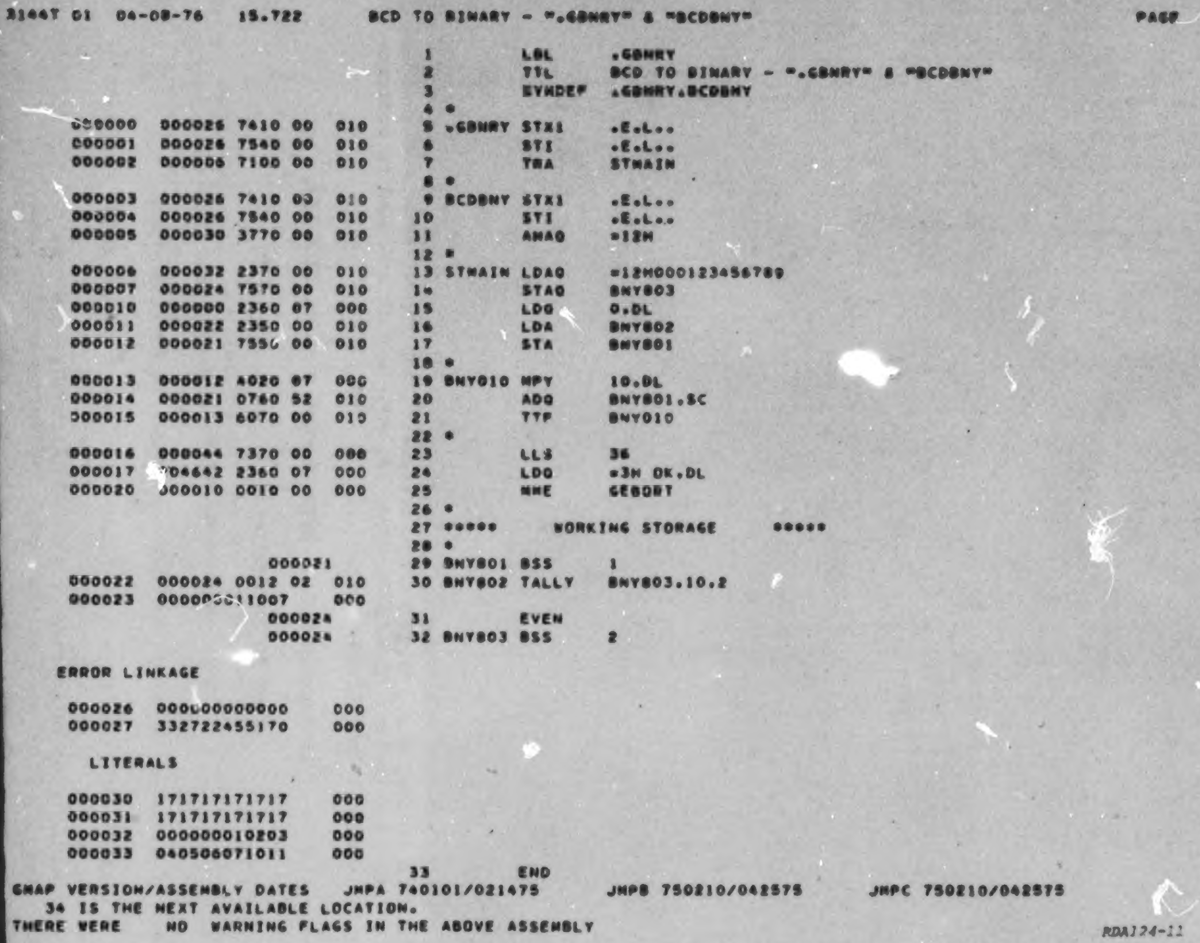
Advantages of Compilers

While they usually don't have the flexibility of assembler-level languages, compiler-level languages have important advantages over assembler-level and machine languages:

1. They have applicability to more than one computer. Although the language specifications are not identical for each computer, the general characteristics are similar enough that usually only minor changes are needed to make a source program compile on different computers.

2. They are easier to learn than machine or assembler language, especially the machine language of very large computer systems.

3. The programming process is speeded up for the programmer once he has learned the syntax of the language.



```
3144T 01  04-08-76   15.722        BCD TO BINARY - ".GBNRY" & "BCDBNY"                              PAGE      2
                                           1         LBL      .GBNRY
                                           2         TTL      BCD TO BINARY - ".GBNRY" & "BCDBNY"
                                           3         SYMDEF   .GBNRY.BCDBNY
                                           4 *
    000000  000026 7410 00  010            5 .GBNRY  STX1     .E.L..
    000001  000026 7540 00  010            6         STI      .E.L..
    000002  000006 7100 00  010            7         TRA      STMAIN
                                           8 *
    000003  000026 7410 00  010            9 BCDBNY  STX1     .E.L..
    000004  000026 7540 00  010           10         STI      .E.L..
    000005  000030 3770 00  010           11         ANAQ     =12H
                                          12 *
    000006  000032 2370 00  010           13 STMAIN  LDAQ     =12H000123456789
    000007  000024 7570 00  010           14         STAQ     BNY803
    000010  000000 2360 07  000           15         LDQ      0.DL
    000011  000022 2350 00  010           16         LDA      BNY802
    000012  000021 7550 00  010           17         STA      BNY801
                                          18 *
    000013  000012 4020 07  000           19 BNY010  MPY      10.DL
    000014  000021 0760 52  010           20         ADQ      BNY801.SC
    000015  000013 6070 00  010           21         TTF      BNY010
                                          22 *
    000016  000044 7370 00  000           23         LLS      36
    000017  004642 2360 07  000           24         LDQ      =3H OK.DL
    000020  000010 0010 00  000           25         MME      GEBORT
                                          26 *
                                          27 *****     WORKING STORAGE     *****
                                          28 *
                   000021                 29 BNY801  BSS      1
    000022  000024 0012 02  010           30 BNY802  TALLY    BNY803.10.2
    000023  000000011007    000
                   000024                 31         EVEN
                   000024                 32 BNY803  BSS      2

   ERROR LINKAGE

    000026  000000000000    000
    000027  332722455170    000

      LITERALS

    000030  171717171717    000
    000031  171717171717    000
    000032  000000010203    000
    000033  040506071011    000
                                          33         END
GMAP VERSION/ASSEMBLY DATES   JMPA 740101/021475      JMPB 750210/042575      JMPC 750210/042575
   34 IS THE NEXT AVAILABLE LOCATION.
THERE WERE    NO  WARNING FLAGS IN THE ABOVE ASSEMBLY
```

RDA124-11

Figure 1-10. Assembler Language Code

4. Communication with others interested in the same problem is much easier with programming statements that read as the problem would commonly be expressed.

5. Because of the standardization, personnel from one installation are able, with little or no retraining, to program for computers at another installation.

COBOL

COBOL (COmmon Business Oriented Language) is the result of an effort to establish a standard language for programming computers to do business-oriented data processing. COBOL is designed for producing source programs that are:

1. Standardized, using standard language elements in standard entry formats within a standard program structure. COBOL endeavors to provide one common language for all computers, regardless of make or model.

2. Easy to understand, because they are written in English. The bulk of every COBOL program is made up of English words in entries that resemble English sentences. Good COBOL programs are easy to read and comprehend for nonprogrammers as well as for programmers.

3. Oriented to business procedures, not to the technology of computing machinery. This makes it possible for business people who are not computer experts to use COBOL.

COBOL uses English language statements as its program instructions. Since these statements can be easily read and understood by people, they provide excellent documentation of the program. For example, the following is a valid COBOL instruction:

```
MULTIPLY HOURS BY HOURLY-WAGE GIVING TOTAL PAY.
```

Verbs such as ADD, MULTIPLY, and PERFORM indicate the action to be taken. Nouns refer to data fields, e.g., in the COBOL statement, ADD AMOUNT TO BALANCE, "AMOUNT" and "BALANCE" are data fields. The COBOL words IF, AND, and OR are used to indicate the testing for specific conditions, e.g.,

```
IF BALANCE EQUALS ZERO OR CODE GREATER THAN 123,

   THEN GO TO ROUTINE-2,

      ELSE ADD 100 TO BALANCE.
```

Some of the advantages of the COBOL language are the following:

1. Reduces Programming Time. Since the COBOL compiler takes over many of the programmer's duties, the programmer does not make as many clerical errors. Usually his errors are in logic or in the use of the COBOL format. In addition, the compiler also provides debugging aids, such as the TRACE, MONITOR, and DUMP commands.

2. Simplifies Change. COBOL's easy-to-read language and standardized format make the programs almost self-documenting. This makes it easier for one programmer to understand and complete or modify another programmer's work, thereby softening the effects of personnel turnover. Equipment changes are also easy to handle. With only minor changes, a COBOL program may be recompiled and run on a new computer system.

3. Produces Efficient Object Programs. The COBOL compiler produces object programs that are nearly equal in efficiency to above-average hand-coded programs. The resultant object programs are usually larger and slower than a hand-coded program, but the difference in programming speed and ease of writing more than compensate for this.

All assemblers and compilers must have certain information before they can generate an object program. This information includes a name for the program to be generated, data file locations (and appropriate hardware names), descriptions of the files and records, specific constants and editing characters, and the particular set of instructions, in proper sequence, needed to process the data.

Unlike most computer languages, the COBOL language divides these various functions into specific categories and allows the programmer to program these functions in four distinct divisions: Identification, Environment, Data, and Procedure.

Identification Division

The Identification Division is for documentation purposes. It contains program name, programmer name, date written, purpose of program, security classification, etc.

Environment Division

The Environment Division is used to furnish the COBOL compiler with information concerning the hardware devices required to produce input-output files.

Data Division

The Data Division is used to describe in detail the input-output formats. It defines data input and output areas, print areas, constants, etc.

Procedure Division

The Procedure Division contains the actual step-by-step instructions of the program. Each statement contains one or more commands, telling the compiler which type of operation is to be performed by the object program. This division is usually written without reference to machine characteristics. Thus, the Procedure Division, if properly written, can be run on any computer which uses the COBOL system.

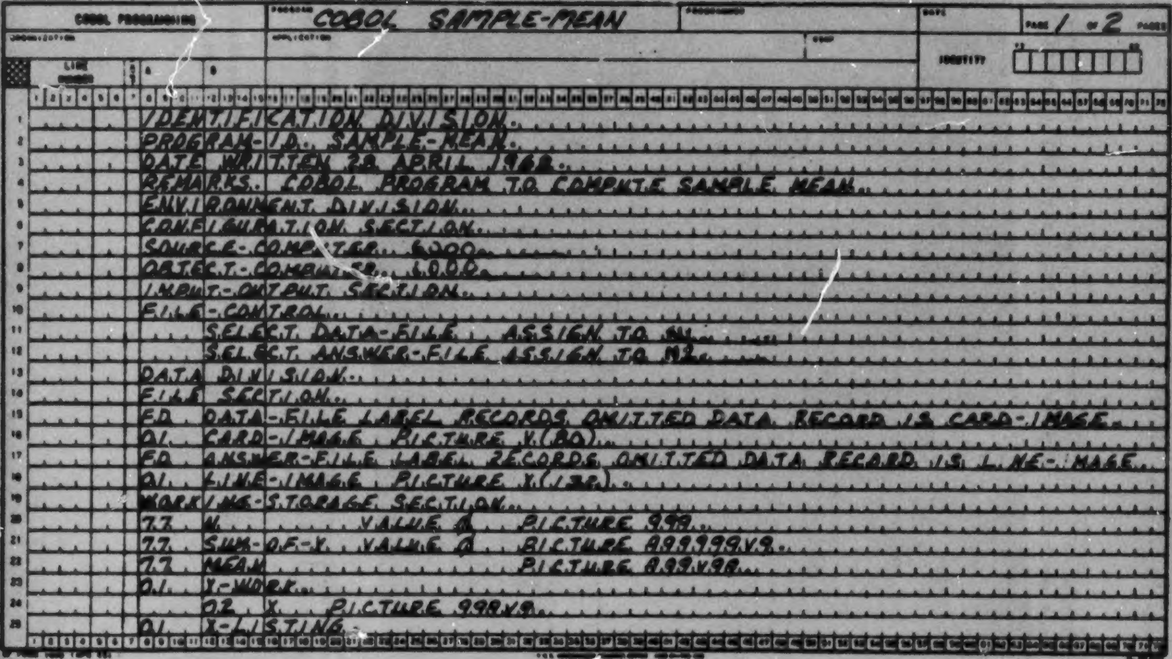
Figure 1-11 shows a sample COBOL program for the Honeywell 6060 computer system.

FORTRAN

FORTRAN (FORmula TRANslation) is the major computational programming language for most United States computing installations. It is the most widely used compiler system in existence.

FORTRAN is a mathematics-oriented language characterized by mathematical statements that indicate the actions to be taken. It permits the programmer to use familiar arithmetic conventions to formulate a program, including such symbols as:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation



COBOL PROGRAMMING

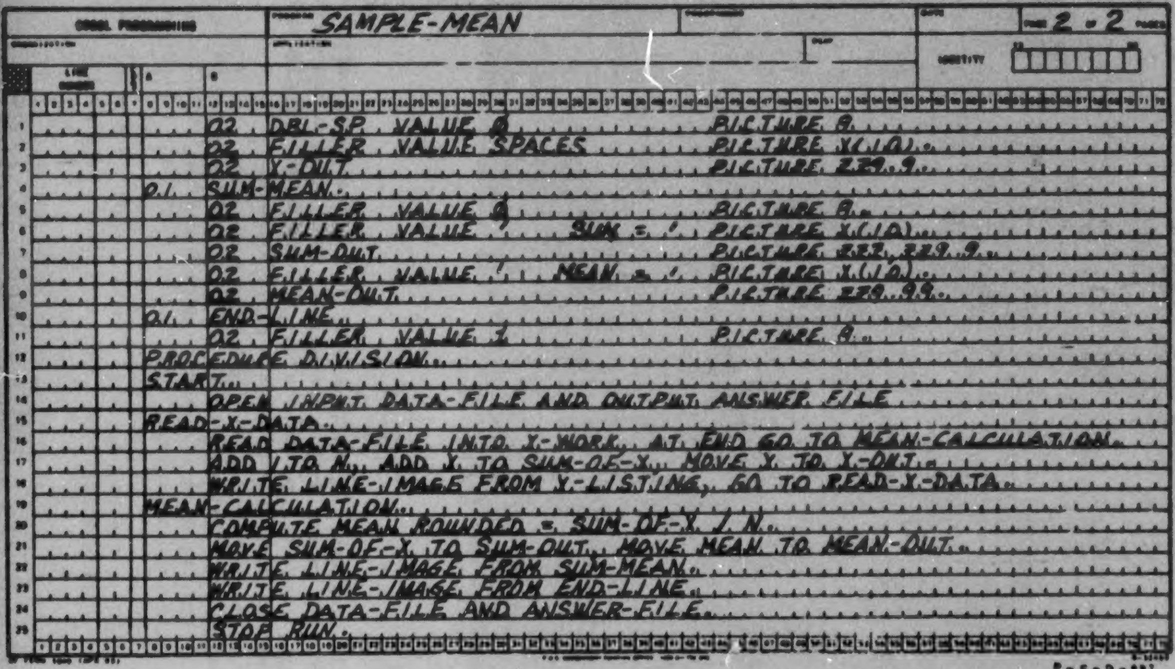
PROGRAM COBOL SAMPLE-MEAN

PAGE 1 OF 2 PAGES

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. SAMPLE-MEAN.
       DATE WRITTEN 28 APRIL 1968.
       REMARKS. COBOL PROGRAM TO COMPUTE SAMPLE MEAN.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. 6000.
       OBJECT-COMPUTER. 6000.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT DATA-FILE  ASSIGN TO M1.
           SELECT ANSWER-FILE ASSIGN TO M2.
       DATA DIVISION.
       FILE SECTION.
       FD  DATA-FILE LABEL RECORDS OMITTED DATA RECORD IS CARD-IMAGE.
       01  CARD-IMAGE  PICTURE X(80).
       FD  ANSWER-FILE LABEL RECORDS OMITTED DATA RECORD IS LINE-IMAGE.
       01  LINE-IMAGE  PICTURE X(132).
       WORKING-STORAGE SECTION.
       77  N              VALUE 0   PICTURE 999.
       77  SUM-OF-X   VALUE 0   PICTURE 999999V9.
       77  MEAN                     PICTURE 999V99.
       01  X-WORK.
           02  X    PICTURE 999V9.
       01  X-LISTING.
```

P-C-0-007
RDA124-12

Figure 1-11. Sample COBOL Program



COBOL PROGRAMMING

PROGRAM SAMPLE-MEAN

PAGE 2 OF 2 PAGES

```
       02  DBL-SP VALUE 0.                 PICTURE 9.
       02  FILLER VALUE SPACES             PICTURE X(10).
       02  X-OUT.                          PICTURE ZZ9.9.
   01  SUM-MEAN.
       02  FILLER VALUE 0                  PICTURE 9.
       02  FILLER VALUE '      SUM = '     PICTURE X(10).
       02  SUM-OUT                         PICTURE ZZZ,ZZ9.9.
       02  FILLER VALUE '     MEAN = '     PICTURE X(10).
       02  MEAN-OUT                        PICTURE ZZ9.99.
   01  END-LINE.
       02  FILLER VALUE 1                  PICTURE 9.
   PROCEDURE DIVISION.
   START.
       OPEN INPUT DATA-FILE AND OUTPUT ANSWER FILE
   READ-X-DATA.
       READ DATA-FILE INTO X-WORK, AT END GO TO MEAN-CALCULATION.
       ADD 1 TO N. ADD X TO SUM-OF-X. MOVE X TO X-OUT.
       WRITE LINE-IMAGE FROM X-LISTING. GO TO READ-X-DATA.
   MEAN-CALCULATION.
       COMPUTE MEAN ROUNDED = SUM-OF-X / N.
       MOVE SUM-OF-X TO SUM-OUT. MOVE MEAN TO MEAN-OUT.
       WRITE LINE-IMAGE FROM SUM-MEAN.
       WRITE LINE-IMAGE FROM END-LINE.
       CLOSE DATA-FILE AND ANSWER-FILE.
       STOP RUN.
```

RDA124-13

Figure 1-11. Sample COBOL Program (Continued)

119

FORTRAN offers two main advantages to the scientific/engineering computer installation: improved communications and greater programming speed.

The FORTRAN programming language is expressed much like mathematical formulas, and is therefore easily learned by most scientists and engineers. These people are able to write their own programs to solve computations, rather than having regular programmers do the work for them. The engineer can be assured of getting the exact results he wants, rather than trying to communicate his needs to a third party. On more complicated problems, the engineer can work closely with the programming staff.

The second advantage, speed, results from FORTRAN's simplified condensed nature. FORTRAN, in effect, is a kind of programming shorthand. This results in shorter, more straightforward programming.

For programs that do not have a large amount of input-output but do have a considerable amount of calculations to be performed, FORTRAN is usually the most efficient programming language. For problems involving calculations, FORTRAN involves less time and cost.

The standardization inherent in FORTRAN programming has several major advantages. A program does not have to be returned to the author for revision, since any programmer with FORTRAN training can understand another programmer's FORTRAN program and expand, revise, or adapt it to another computer system. This flexibility can cut costs during normal operating conditions and offers even greater savings when a new system is installed and software conversions must be made.

Figure 1-12 is an example of a FORTRAN program.

ALGOL

ALGOL (ALGOrithmetic Language) is the result of an international effort, parallel to the development of FORTRAN in the United States. It is a science and mathematically oriented program language, characterized by the use of algorithms (an algorithm is a fixed, step-by-step procedure that accomplishes a given result).

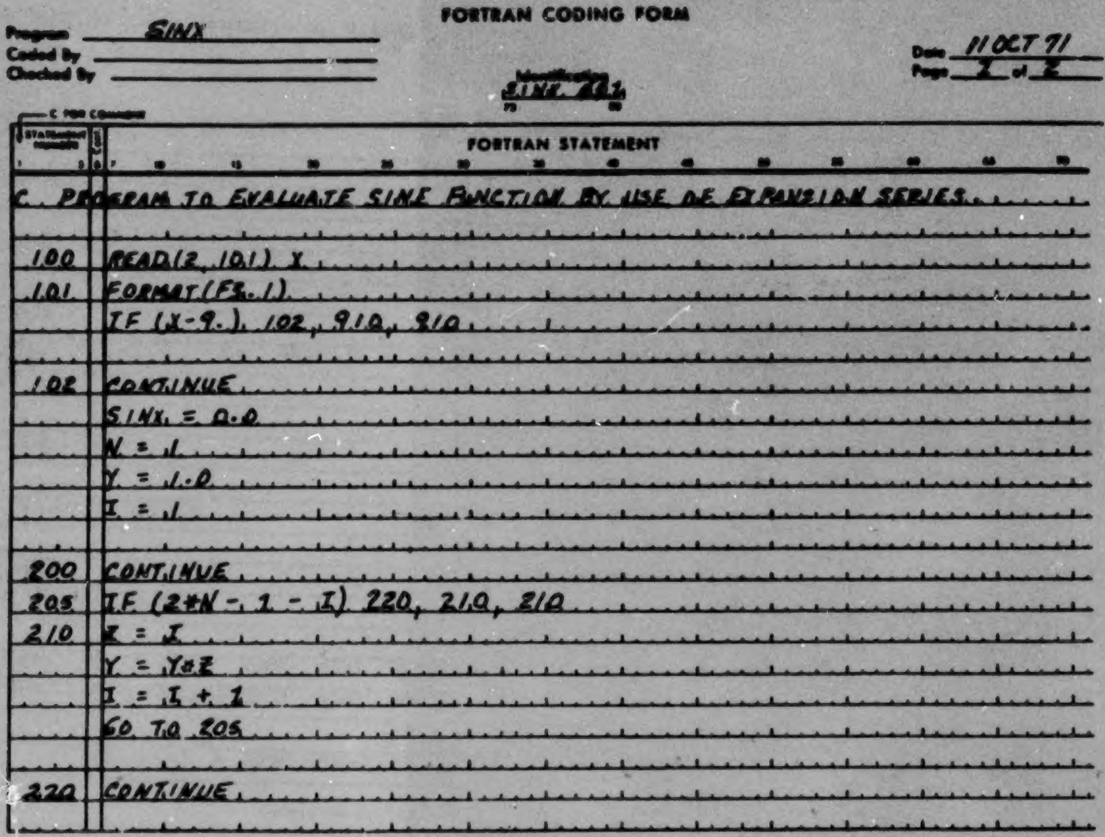
ALGOL is made up of mathematical statements which express actions to be taken. It permits the programmer to use familiar arithmetic conventions to formulate a program: + addition; - subtraction; X multiplication; / division; * exponentiation.

Like FORTRAN, ALGOL allows the mathematician or scientist to concentrate on the problem and not be overly concerned with the computer's machine language.

PL/1

In the past, throughout the data processing field, certain computers were generally identified with a particular field of activity, either scientific or commercial. Programming languages were specialized in the same way: FORTRAN was developed for scientific programming and COBOL for commercial programming.

Now, however, computing systems are designed for a broader range of activity. The new computers are faster and more powerful. They serve the scientific and commercial programmer equally well, and they provide facilities for many new programming techniques. None of the older languages can take advantage of all the power of the new computers, and more and more computer installations are handling both scientific and commercial programming.



FORTRAN CODING FORM

Program SINX
Coded By
Checked By

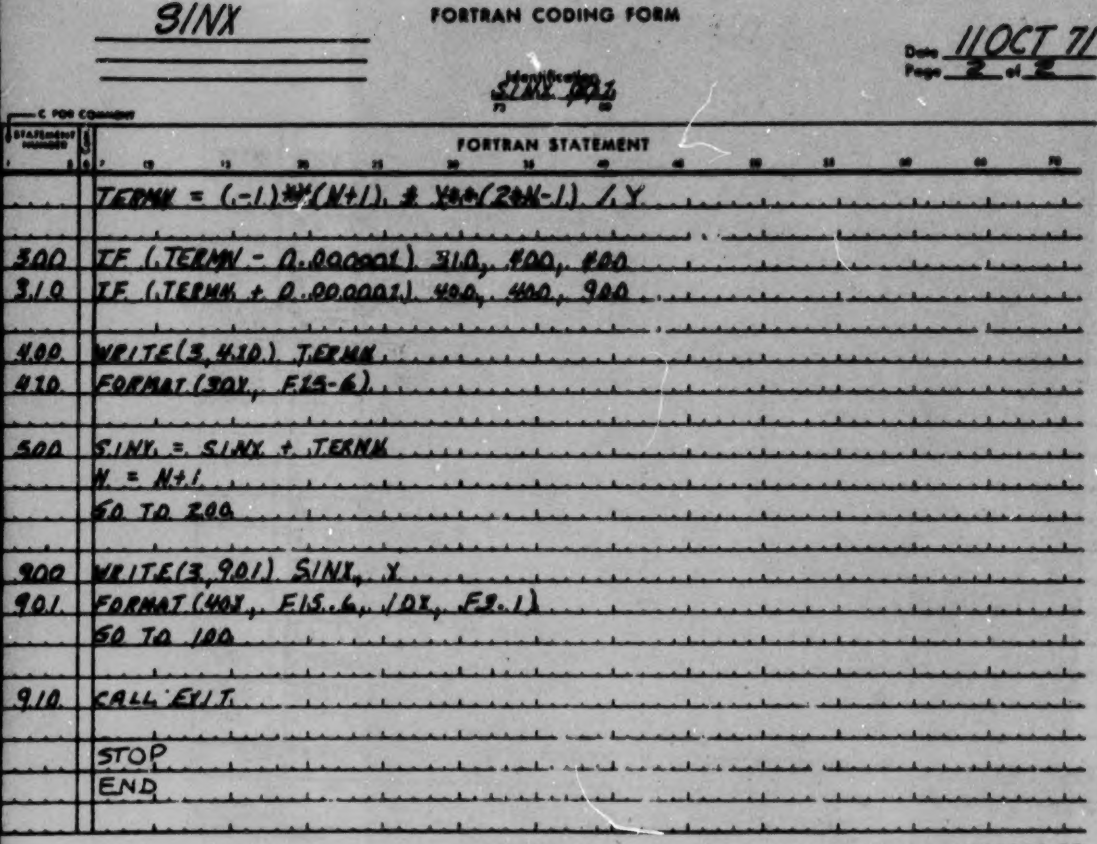
Identification SINX 221

Date 11 OCT 71
Page 1 of 2

```
C PROGRAM TO EVALUATE SINE FUNCTION BY USE OF EXPANSION SERIES.

100   READ(2,101) X
101   FORMAT(F3.1)
      IF (X-9.) 102, 910, 910

102   CONTINUE
      SINX = 0.0
      N = 1
      Y = 1.0
      I = 1

200   CONTINUE
205   IF (2*N - 1 - I) 220, 210, 210
210   Z = I
      Y = Y*Z
      I = I + 1
      GO TO 205

220   CONTINUE
```

RDA124-14

Figure 1-12. Example of a FORTRAN Program



SINX

FORTRAN CODING FORM

Date 11OCT 71
Page 2 of 2

Identification SINX 001

C FOR COMMENT

```
      TERMN = (-1)**(N+1) * Y**(2*N-1) / Y

300   IF (TERMN - 0.000001) 310, 400, 500
310   IF (TERMN + 0.000001) 400, 400, 900

400   WRITE(3,410) TERMN
410   FORMAT(30X, F15-6)

500   SINX = SINX + TERMN
      N = N+1
      GO TO 200

900   WRITE(3,901) SINX, X
901   FORMAT(40X, E15.6, 10X, F9.1)
      GO TO 100

910   CALL EXIT

      STOP
      END
```

RDA124-15

Figure 1-12. Example of a FORTRAN Program (Continued)

PL/1 (Programming Language/1) is a multipurpose programming language that can be used by both commercial and scientific programmers to handle all of their programming and to give them the widest range of control over the computer.

PL/1 has been designed so that any programmer, no matter how brief or extensive his experience, can use it easily at his own level. It is simple for the beginning programmer; it is powerful for the experienced one. A programmer need not know everything about PL/1 to be able to use it. An experienced programmer can use PL/1 to specify almost every detail of every step of a highly complicated program. A beginner can take advantage of the many automatic features of the language to do much of his work for him.

REPORT PROGRAM GENERATOR

The purpose of the Report Program Generator (RPG) and its resultant object program is to enable the user to obtain comprehensive reports from existing files with a minimum of time involved in source coding.

The object program, which is produced by the Report Program Generator from source coding on appropriate forms, is more efficient than most programs created using the assembler and compiler languages alone.

Source coding for the Report Program Generator is on six types of punched cards, divided into three divisions:

1. The Environment Division - Contains program ID, options, file and record description, printer control, and accumulator descriptions.
2. The Data Division - Contains details on constants, working storage, etc.
3. The Procedure Division - Contains the step-by-step procedures for solving the problem.

SIMSCRIPT

SIMSCRIPT (SIMulation) is used to simulate, in a computer, the operation of a physical system, such as manufacturing, logistics, transportation, or economics. Simulation is essentially a technique that involves setting up a model of a real situation and then performing results on the model.

The program is used to test the performance of alternate physical systems obtained when different sets of decisions are used under various conditions.

For example, when a transportation system is simulated, the computer is told the physical structure of the system, e.g., the number of trucks, number of drivers. The computer is also told the rules that determine how the system will operate, e.g., the deliveries to be made, the hours a driver can work, destinations.

With such information, the computer simulates the deliveries to be made by each truck and driver. Performance of the system is based on factors such as truck utilization, labor utilization, promptness of delivery.

The simulated system can be operated any number of times, varying the quantity of goods to be delivered, the decision rules to be followed, or the configuration of the system itself (the number of trucks and drivers). By simulating alternate systems under various conditions, management can devise the system best suited for their needs.

JOVIAL

JOVIAL (Jules Own Version of an International Algebraic Language) is a machine-independent, general-purpose programming language, answering the need for a common standard of communications between users of many different computers. As a common programming language, JOVIAL serves both as a means of communicating information processing methods between people and as a means of realizing a stated process on a number of different computers. JOVIAL is considered a standard language for command control applications.

125

CHAPTER 2

COMPUTER MATHEMATICS

OBJECTIVES

When you have completed this chapter, you should be able to:

1. Identify the distinguishing features of numbering systems using base 2, 8, and 10.

2. Convert any number expressed in one of these three numbering systems to its equivalent in any of the other two.

3. Add and subtract numbers in binary and octal numbering systems.

4. Complement binary and octal numbers.

INFORMATION

Before beginning the study of digital computer programming, it is necessary to study the numbering system on which the language of the computer is based. This basic numbering system is the binary system. However, the binary system will not be the only numbering system discussed in this chapter; octal and decimal systems will also be studied. The octal system will be studied because binary numbers can be expressed and manipulated more conveniently in equivalent octal form. The decimal system will be studied so that the fundamentals that apply to all numbering systems can be learned in the content of a familiar system.

NUMBER SYSTEMS

Decimal System (Positional and Symbolic Values of Digits)

The number of different symbols used in our numbering system determines the base of that system. The base of the decimal system is 10, since it uses as symbols the 10 Arabic numerals--0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. These 10 symbols have a commonly known order or sequence which enables us to count and to indicate quantity with them. They may be used alone or in combination to express quantities or identities.

By using 10 symbols and given significance to their relative position with respect to a decimal point, we can combine the symbols in a number of ways to describe any value we choose. Considering a whole number (i.e., no decimal fraction), the extreme right digit is multiplied by one and is called the Least Significant Digit (LSD). The next digit to the left of the LSD is multiplied by 10, and the next left one is multiplied by 100. In decimal notation, adjacent digits have a 10-to-1 ratio, increasing from the LSD to the Most Significant Digit (MSD). Examination of an example from the decimal system reveals that the positions have the following significance: The number 4358 means four thousands plus three hundreds plus five tens plus eight ones, or:

$$(4 \times 1000) + (3 \times 100) + (5 \times 10) + (8 \times 1)$$

or

$$(4 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (8 \times 10^0)$$

The following chart (chart 2-1) should be studied to firmly establish in your mind the value of each digit in the decimal system.

Chart 2-1

Hundred thousands	Ten thousands	Thousands	Hundreds	Tens	Ones
10^5 X (100000)	10^4 X (10000)	10^3 X (1000)	10^2 X (100)	10^1 X (10)	10^0 X (1)

Each position to the left of the decimal point represents a positive power of 10, with the power increasing from each position to the next left position. Positions to the right of the decimal point represent negative powers of 10, with the power increasing in a negative direction from each position to the next right position. The negative power of a number is the number of times the reciprocal of the number is multiplied by itself. Examine the decimal number 65.421. The expression is equivalent to:

$$(6 \text{ X } 10^1) + (5 \text{ X } 10^0) + (4 \text{ X } 10^{-1}) + (2 \text{ X } 10^{-2}) + (1 \text{ X } 10^{-3})$$

or

$$6(10) + 5(1) + 4(0.1) + 2(.01) + 1(.001)$$

Binary System

As indicated by the prefix BI, this system utilizes only two symbols, "0" and "1." Therefore, the binary system has a base of 2, and all numbers in this system are combinations of the various powers of 2. The general rules and mathematical functions of the decimal system apply in the binary or base two system although the columns in binary are valued at base two significance. The following chart (chart 2-2) shows the decimal equivalent for each position of a binary number.

Chart 2-2

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

Comparing the decimal system to the binary system will clarify the idea. See chart 2-3 on the following page.

Notice that in binary each progressing column is exactly double the value of the preceding column. The base of the binary system is 2; consequently, any binary number can be expressed as a sum of the powers of 2. The value of a binary digit (bit) is determined by its position in relation to the binary point (similar to a decimal point).

Chart 2-3

Decimal No.		Power of Ten	Binary No.		Power of Two		Decimal Equivalent
1	=	10^0	1	=	2^0	=	1
10	=	10^1	10	=	2^1	=	2
100	=	10^2	100	=	2^2	=	4
1000	=	10^3	1000	=	2^3	=	8
10000	=	10^4	10000	=	2^4	=	16

It is simple enough to relate binary numbers to their decimal equivalents. For example, in order to relate the binary number 10011 to its decimal equivalent, write down the powers of two and set the binary number under them (see chart 2-4 below).

Chart 2-4

Powers of Two:	2^4	2^3	2^2	2^1	2^0
Binary Number:	1	0	0	1	1
Decimal Equivalent:	16	+ 0	+ 0	+ 2	+ 1 = 19

The decimal equivalent of this number is shown in the following equation:

$$(1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

In other words, the binary number 10011 is equal to the decimal number 19.

The following chart (chart 2-5) is a listing of some binary numbers and their decimal equivalents.

Chart 2-5

Binary	Decimal	Binary	Decimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Each position to the left of the binary point represents a positive power of 2, with the power increasing from each position to the next left position. Positions to the right of the binary point represent negative powers of 2, with the power increasing in a negative direction from each position to the next right position. Examine the binary number 101.1111. The expression is equivalent to:

$$(1 \text{ X } 2^2) + (0 \text{ X } 2^1) + (1 \text{ X } 2^0) + (1 \text{ X } 2^{-1}) + (1 \text{ X } 2^{-2}) + (1 \text{ X } 2^{-3}) + (1 \text{ X } 2^{-4})$$

or

$$4 + 0 + 1 + 1/2 + 1/4 + 1/8 + 1/16$$

Octal System

The octal number system is similar to the decimal and binary systems. The only difference in the three systems is caused by the differences in base. Whereas the base for the decimal system is "10" and the base for the binary system is "2," the base of the octal system is "8." Therefore, the only permissible symbols in the octal system are 0, 1, 2, 3, 4, 5, 6, and 7. Notice that the decimal symbols "8" and "9" are NOT used.

In the octal system the value of a digit is determined by its position relative to an octal point (similar to a binary or decimal point). You recall that the octal system is based on powers of 8 and, just as we have seen in the other systems, we can think of the digits in an octal number as the coefficients of the powers of 8. These coefficients determine how many times each power of the base number 8 is to be added to the sum. The following chart (chart 2-6) shows the decimal equivalent for each position of an octal number.

Chart 2-6

8^6	8^5	8^4	8^3	8^2	8^1	8^0
262144	32768	4096	512	64	8	1

The decimal equivalent of octal number 14025 is shown below in chart 2-7.

Chart 2-7

Powers of Eight:	8^4	8^3	8^2	8^1	8^0	
Octal Number:	1	4	0	2	5	
Decimal Equivalent:	4096 +	2048 +	0 +	16 +	5	= $6165_{(10)}$

Each position to the left of the octal point represents a positive power of 8 with the power increasing from each position to the next left position. Positions to the right of the octal point represent negative powers of 8 with the power increasing in a negative direction from each position to the next right position. Examine the octal number 76.345. The expression is equivalent to:

$$(7 \text{ X } 8^1) + (6 \text{ X } 8^0) + (3 \text{ X } 8^{-1}) + (4 \text{ X } 8^{-2}) + (5 \text{ X } 8^{-3})$$

or

$$7(8) + 6(1) + 3(1/8) + 4(1/64) + 5(1/512)$$

The octal number system is useful as a form of shorthand for the binary system. The relationship between the two systems can be stated as follows: Since "8" is the third power of "2," three places in binary notation correspond to one place in octal notation; consequently, three binary digits can be represented by one octal digit. The correspondence can be summarized as follows (chart 2-8):

Chart 2-8

BINARY is equivalent to	OCTAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

The following diagram illustrates the value of the octal system as a form of binary shorthand.

110101011111 = 110 101 011 111

6 5 3 7

The use of eight symbols (octonary), rather than two symbols (binary), provides a numbered language which is easier to read and write than binary; thus, it decreases the likelihood of a programmer error. It is obvious that the octal system is a good working system for programmers because octal notation is much shorter than binary and because conversion between the two systems is relatively simple.

By simple inspection, there is no way of knowing whether the number 1011 is an octal, a decimal, or a binary number. Subscripts are normally used to designate the type of number system. For example, 1011_8 or $1011_{(8)}$ indicates that the number is written in octal; accordingly, the subscripts "2" and "10" indicate that the number is written in binary and decimal, respectively.

NUMBER CONVERSION

The preparation of information for digital computers involves the use of the decimal, octal, and binary numbering systems. Consequently, conversion from one numbering system to another is often necessary.

Decimal to Binary Conversion

The division method is used to convert whole decimal numbers to binary numbers. To convert the decimal number to binary, divide by two. The quotient obtained is again divided by two, and so on. The binary coefficients are indicated by a "1" each time the division results in a remainder of 1 and by a "0" each time no remainder is obtained.

The first binary bit obtained is the least significant bit (LSB), and the last bit obtained is the most significant bit (MSB) of the binary expression. Study the following conversion of $123_{(10)}$ to $N_{(2)}$:

1st division ——►	2)$\underline{123}$		
2nd division	2)$\underline{61}$	1 ◄——	1st remainder (LSB)
3rd division	2)$\underline{30}$	1	2nd remainder
4th division	2)$\underline{15}$	0	3rd remainder
5th division	2)$\underline{7}$	1	4th remainder
6th division	2)$\underline{3}$	1	5th remainder
7th division ——►	2)$\underline{1}$	1	6th remainder
	0	1 ◄——	final remainder (MSB)

$123_{(10)}$ equals $1111011_{(2)}$

The following method is used to convert a decimal fraction to its binary equivalent. The fraction is repeatedly multiplied by 2. Each time that the multiplication generates a product having the whole number 1, a 1 is entered as a coefficient of the equivalent binary expression. (The 1 is then dropped.) Each time that the product of the multiplication is a fraction, the 0 is entered into the binary sequence of coefficients. This process is continued until a product of 1.0 is obtained, or until the desired number of bits have been obtained. The first bit obtained is the MSB of the binary fraction. Study the following conversions to the equivalent binary numbers:

$$\begin{array}{r} .875 \\ \underline{\times\ \ 2} \\ 1.750 \\ \downarrow \\ 1 \\ \text{(MSB)} \end{array} \qquad \begin{array}{r} .75 \\ \underline{\times\ 2} \\ 1.50 \\ \downarrow \\ 1 \\ \ \end{array} \qquad \begin{array}{r} .5 \\ \underline{\times\ 2} \\ 1.0 \\ \downarrow \\ 1 \\ \text{(LSB)} \end{array} \ = .111$$

$$.875_{(10)} = .111_{(2)}$$

$$\begin{array}{r} .0625 \\ \underline{\times\ \ \ \ 2} \\ 0.1250 \\ \downarrow \\ 0 \\ \text{(MSB)} \end{array} \qquad \begin{array}{r} .125 \\ \underline{\times\ \ \ 2} \\ 0.250 \\ \downarrow \\ 0 \\ \ \end{array} \qquad \begin{array}{r} .25 \\ \underline{\times\ 2} \\ 0.50 \\ \downarrow \\ 0 \\ \ \end{array} \qquad \begin{array}{r} .5 \\ \underline{\times\ 2} \\ 1.0 \\ \downarrow \\ 1 \\ \text{(LSB)} \end{array} \ = .0001$$

Binary to Decimal Conversion

The conversion of an expression in binary form to its decimal equivalent is performed by using the expansion method. The number $101101_{(2)}$ is a representation of:

$$(1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

which is equal to

$$(1 \times 32) + (0 \times 16) + (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$$

Which is equal to

$$32 + 0 + 8 + 4 + 0 + 1$$

or $45_{(10)}$

The binary fraction .1101 is actually a representation of

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

which is equal to

$$(1 \times .5) + (1 \times .25) + (0 \times .125) + (1 \times .0625)$$

which is equal to

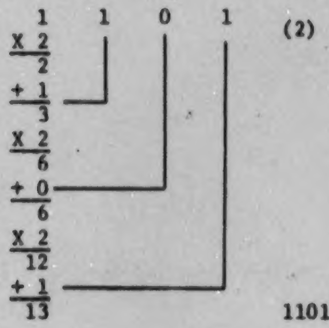
$$.5 + .25 + 0 + .0625 \text{ or } .8125_{(10)}.$$

Study the following example of a conversion of $10011.01_{(2)}$ to $N_{(10)}$ (chart 2-9).

Chart 2-9

Powers of Two:	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
Binary Number:	1	0	0	1	1	.0	1
Decimal Equivalent:	16 +	0 +	0 +	2 +	1 +	.0 +	.25 = 19.25

Another method of converting a binary number is by multiplying the binary digits by 2 starting from the MSD first, then adding the next digit to the right to the product. Repeat this process until all digits have been added. (Note that the MSD is never added.)



$1101_{(2)} = 13_{(10)}$

This method is to be used on integer digits only; use direct look-up on power of 2 chart for fractional digits.

Decimal to Octal Conversion

The division method is used to convert whole decimal numbers to octal numbers. To convert a decimal number to octal, divide by 8, then divide the quotient by 8; continue the division until the quotient is zero. The remainder obtained by the repeated divisions will be the coefficient to the octal number. The first remainder is the octal LSD, and the final remainder is the MSD. Study the following conversion of $3844_{(10)}$ to $N_{(8)}$.

1st division ⟶	8)3844		
2nd division	8) 480	4	⟵ 1st remainder (LSD)
3rd division	8) 60	0	2nd remainder
4th division ⟶	8) 7	4	3rd remainder
	0	7	⟵ final remainder (MSD)

$3844_{(10)}$ is equivalent to $7404_{(8)}$

To convert a decimal fraction to an octal fraction, multiply the decimal fraction by 8. The whole number portion of the product thus obtained is the MSD of the octal fraction. The fractional portion of the product is multiplied by 8 and the whole number of the product is zero, or until the desired number of octal digits have been obtained. Study the following conversion of $.384_{(10)}$ to its octal equivalent (5 octal places).

.384	.072	.576	.608	.864
X 8	X 8	X 8	X 8	X 8
3.072	0.576	4.608	4.864	6.912
↓	↓	↓	↓	↓
3 (MSD)	0	4	4	6 (LSD)

$.384_{(10)}$ is equivalent to $.30446_{(8)}$.

Octal to Decimal Conversion

The conversion of an octal number to its decimal equivalent is performed by using the expansion method. The number $10247.2_{(8)}$ is a representation of:

$$(1 \times 8^4) + (0 \times 8^3) + (2 \times 8^2) + (4 \times 8^1) + (7 \times 8^0) + (2 \times 8^{-1})$$

which is equal to

$$(1 \times 4096) + (0 \times 512) + (2 \times 64) + (4 \times 8) + (7 \times 1) + (2 \times .125)$$

which is equal to

$$4096 + 0 + 128 + 32 + 7. + .250$$

or

$$4263.25_{(10)}.$$

The multiplication method of converting octal to decimal is performed by multiplying the octal digits by 8 starting with the MSD first, then add the next digit to the right to the product unit until all digits have been added.

```
   7   3   4   3   (8)
X  8
  56
+  3
  59
X  8
 472
+  4
 476
X  8
3808
+  3
3811
```

$7343_{(8)} = 3811_{(10)}$

This method is to be used on integer digits only; use direct look-up power of 8 chart for fractional digits.

Octal to Binary Conversion

Conversion from octal to binary is accomplished by the inspection method. By replacing each digit in the octal number with a group of three binary digits (bits), octal number 3752.01 is converted as follows:

```
 3   7   5   2  .  0   1
011 111 101 010 . 000 001
```

Therefore, $3752.01_{(8)}$ is equivalent to $011\ 111\ 101\ 010.000\ 001_{(2)}$.

Binary to Octal Conversion

The method of conversion from binary to octal is "conversion by inspection." To make this conversion, arrange the binary bits in groups of three, beginning at the binary point and proceeding to the left (for whole numbers) and to the right (for fractions). If either the extreme left or right groups contain fewer than three bits, fill out the groups with "0's." Study the following conversion of $11010111110.000101_{(2)}$ to $N_{(8)}$.

```
011 010 111 110 . 000 101
 3   2   7   6  .  0   5
```

$11010111110.000101_{(2)}$ is equivalent to $3276.05_{(8)}$.

Summary of Conversions

Decimal to Binary (INTEGERS). Divide by 2 and save remainders (LSD first).

Decimal to Binary (FRACTIONS). Multiply by 2 and save overflow (MSD first).

Binary to Decimal (INTEGERS). Multiply MSD by 2 and add next digit to the right to product; repeat until all digits are used up.

Binary to Decimal (FRACTIONS). Look up on a negative power of 2 table.

When converting from DECIMAL to OCTAL and OCTAL to DECIMAL, use the same sequence as used in binary, except for octal use 8 as the underlined number.

Binary to Octal. Starting with the radix point, group the binary digits in three's and express as octal.

INTEGERS ← . → FRACTIONS

Octal to Binary. Keep the radix point and express each octal digit as its equivalent three binary digits.

ARITHMETIC OPERATIONS

The basic rules for addition and subtraction are the same for any numbering system--decimal, octal, or binary. The rules are as follows:

Addition

1. When adding like signs, perform a straight addition and retain the sign.

2. When adding unlike signs, subtract the smaller from the larger and retain the sign of the larger value.

Subtraction

When subtracting (either like or unlike signs), change the sign of the subtrahend and proceed according to the rules for addition.

Decimal Addition			
	9	-16	54
	4	-37	-32
	13	-53	22
Decimal Subtraction	9	-76	61
	(-) -2	(-) -42	(-) 34
	11	-34	27

When a carry of 1 is required to complete decimal addition, or when a borrow of 1 is required to complete decimal subtraction, notice what value this 1 represents in the decimal numbering system. It represents one times the radix of the decimal system (i.e., 1 X 10 or 10). This fact, often overlooked in familiar decimal addition and subtraction, must be understood in order to correctly perform addition and subtraction in the octal or binary numbering systems.

The 1 which may be carried or borrowed in octal addition and subtraction represents one times the radix of the octal numbering system (i.e., 1 X 8 or 8). Likewise, the 1 which may be carried or borrowed in binary addition and subtraction represents one times the radix of the binary system (i.e., 1 X 2 or 2). Keep these facts in mind for correct octal or binary addition and subtraction. (See Appendix F for octal and binary tables.)

Octal Addition

$$\begin{tikzcd} 16_{(8)} \arrow[dr] & -47_{(8)} \\ \underline{-47_{(8)}} \arrow[ur] & \underline{16_{(8)}} \\ & -31_{(8)} \end{tikzcd}$$

$$\begin{array}{r} 7_{(8)} \\ \underline{7_{(8)}} \\ 16_{(8)} \end{array} \qquad \begin{array}{r} -471_{(8)} \\ \underline{-234_{(8)}} \\ -725_{(8)} \end{array}$$

Octal Subtraction

$$\begin{array}{rr} & 7_{(8)} \\ (-) & \underline{5_{(8)}} \\ & 2_{(8)} \end{array} \qquad \begin{array}{rr} & -26_{(8)} \\ (-) & \underline{7_{(8)}} \\ & -17_{(8)} \end{array} \qquad \begin{array}{rr} & -6543_{(8)} \\ (-) & \underline{-4672_{(8)}} \\ & -1651_{(8)} \end{array} \qquad \begin{array}{rr} & 65_{(8)} \\ (-) & \underline{16_{(8)}} \\ & 47_{(8)} \end{array}$$

Binary Addition

$$\begin{array}{r} 1_{(2)} \\ \underline{0_{(2)}} \\ 1_{(2)} \end{array} \qquad \begin{array}{r} 1_{(2)} \\ \underline{1_{(2)}} \\ 10_{(2)} \end{array} \qquad \begin{array}{r} 110_{(2)} \\ \underline{-011_{(2)}} \\ 11_{(2)} \end{array} \qquad \begin{array}{r} -11_{(2)} \\ \underline{-10_{(2)}} \\ -101_{(2)} \end{array}$$

Binary Subtraction

$$\begin{array}{rr} & 111_{(2)} \\ (-) & \underline{101_{(2)}} \\ & 10_{(2)} \end{array}$$

$$\begin{tikzcd} & 110_{(2)} \arrow[dr] & -111_{(2)} \\ (-) & \underline{111_{(2)}} \arrow[ur] & \underline{110_{(2)}} \\ & & -\ 1_{(2)} \end{tikzcd}$$

$$\begin{array}{rr} & -1001_{(2)} \\ (-) & \underline{-0111_{(2)}} \\ & -\ 10_{(2)} \end{array}$$

Radix Complement

The radix complement of a number is the inversion of a numeric value, derived by subtracting the numeric value from the number of counting combinations possible by the numeric value's positional power. This definition of radix complement is not as difficult as it sounds, as examples will show.

Eight's Complement (Radix Complement of Octal Numbers)

$$562_{(8)} \qquad \begin{array}{rr} & 1000_{(8)} \\ (-) & \underline{562_{(8)}} \\ & 216_{(8)} \end{array} \quad \text{Eight's Complement}$$

$$63_{(8)} \qquad \begin{array}{rr} & 100_{(8)} \\ (-) & \underline{63_{(8)}} \\ & 15_{(8)} \end{array} \quad \text{Eight's Complement}$$

$$4162_{(8)} \qquad \begin{array}{r} 10000_{(8)} \\ (-)\ \underline{4162_{(8)}} \\ 3616_{(8)} \end{array} \qquad \text{Eight's Complement}$$

Two's Complement (Radix Complement of Binary Numbers)

$$101_{(2)} \qquad \begin{array}{r} 1000_{(2)} \\ (-)\ \underline{101_{(2)}} \\ 011_{(2)} \end{array} \qquad \text{Two's Complement}$$

$$110101_{(2)} \qquad \begin{array}{r} 1000000_{(2)} \\ (-)\ \underline{110101_{(2)}} \\ 001011_{(2)} \end{array} \qquad \text{Two's Complement}$$

Note that the Honeywell 6000 series computers use the radix complement of binary numbers to perform subtraction operations inside the machine.

CHAPTER 3

CONCEPTS OF DATA DESCRIPTION

OBJECTIVE

After completion of this chapter, you should be able to demonstrate an understanding of data representation and organization by defining the following terms: bit, character, word, field, logical record, physical record, and file.

INTRODUCTION

This chapter will give you a general idea of how data can be represented and organized by a computer. This idea will be utilized in the flowchart problems in both Chapter 4 and your workbook.

INFORMATION

In any application, related data is grouped in certain ways. Even the largest mass of data can be treated as a single item. Thus, a dictionary can be considered a book (single item) or it can be considered a series of chapters (series of single items) or a series of definitions, words, or characters (another series of single items). In each case, there is some pattern by which the items are grouped. Likewise, bodies of related information are grouped at all levels of computer usage.

Many terms will be defined in this section so that data representation and organization may be explained. These terms are as follows:

1. Bit - A binary digit which has either a 0 or 1 value.

2. Character - One symbol of a set of elementary symbols, such as those corresponding to the keys of a typewriter.

3. Word - The number of characters which occupy one storage location in the memory element of a computer (treated as one unit of transfer).

4. Field - A group of characters which are related (specified by the programmer).

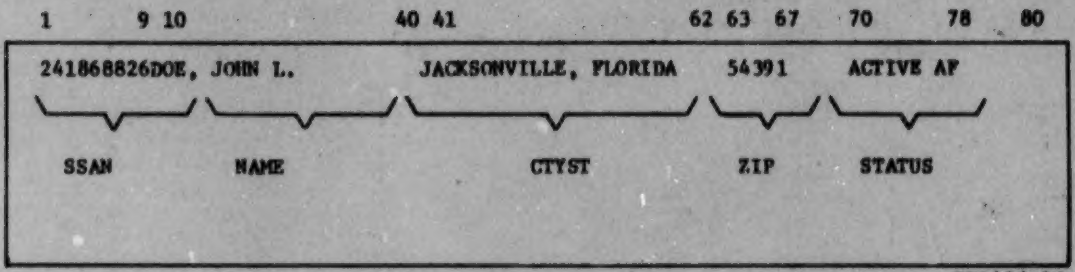
5. Logical record (record) - A group of related fields which are treated as a unit by the programmer.

6. Physical record - The number of logical records that are read at one memory access of a computer (makes I/O more efficient).

7. File - A number of logical records (which normally have the same format).

Now, to add meaning to the terms defined above, look at some of them from a programmer's viewpoint. A programmer's file may be one of many files on some mass storage device (i.e., magnetic tape or disk). This file is made up of a specified number of records which, for simple explanation, will be formatted the same. Each record is broken up into fields which, as you recall, are simply groups of characters/symbols (i.e., DOG, NAME, 1245, etc.).

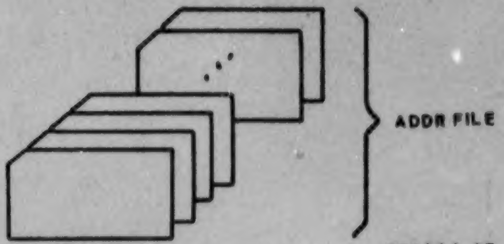
Graphically, a sample 80-character logical record might look like figure 3-1 on the following page.



RDA124-16

Figure 3-1. Sample Logical Record

Notice that the fields in the sample record have symbolic names associated with them. Most compiler languages (i.e., FORTRAN, COBOL, etc.) allow the programmer to reference fields by using the name of that field. Incidentally, the kind of characters/symbols allowed in a particular field would be specified by the programmer setting up the record format for a file. A file with records formatted like figure 3-1 might look graphically like figure 3-2.



RDA124-17

Figure 3-2. Sample Card File

The file represented in figure 3-2 might be an address file for a student squadron. This file might have the symbolic name ADDR and would be one of many files in a computer.

For records of a file to be ordered, one of its fields is used as a key field. This would be the SSAN (Social Security Number) field for the records in ADDR. ADDR could be sorted in ascending or descending order based on its key field SSAN.

Now look at some of the previously defined terms from a machine viewpoint. Computers read and transfer data in units called words, which you remember are the number of characters which occupy one storage location in the memory element of a computer. The Honeywell 6000 series computers use a word which is 36 bits wide. The number of characters/symbols that can be represented depends on which machine code (i.e., BCD, ASCII, etc.) is used to represent an individual character/symbol. If Binary Coded Decimal (BCD) is used, the Honeywell 36-bit word can represent six BCD characters. This is because each character/symbol in BCD can be represented by six binary numbers (see figure 3-3 and refer to Appendix E).

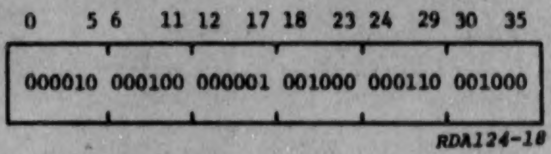


Figure 3-3. Honeywell 6000 Word

The binary numbers (0 or 1) in the figure 3-3 word represent six decimal digits (241868). It follows that 80 Honeywell words would be needed to store each 80-character record of the ADDR file in figure 3-2.

To facilitate more efficient input, a computer may retrieve more than one logical record in a read to a file in memory. The number of logical records read at one time is called a physical record. In the sample card file of figure 3-2, a physical record might be equal to ten logical records.

So far this section has emphasized how related data can be grouped, specifically as records in files. Data may also be used as single items. Essentially, a single item of data would be a field of characters/symbols in memory which is not a field within a record of some file. These single fields are also normally referred to in programming languages by using a symbolic name associated with them. ENDD might be the symbolic name of a single item of data which indicates the number of records in a particular file in memory. KNT might be the symbolic name for a single item used as a counter to keep up with the number of records read. The symbolic names of data fields (single or part of a record) are often called variables by many programming languages.

This section has defined some basic terms of data representation and organization that give you a general idea of how data looks to a programmer and the machine itself. Keep this information in mind when you are solving the flow-charting problems in Chapter 4.

CHAPTER 4

PROBLEM SOLVING AND FLOWCHARTING

OBJECTIVES

After completion of this chapter, you should be able to:

1. Identify the four methods of solving a problem.

2. Identify what is a problem, and define the problem in terms of an algorithm.

3. Define structured programming and list its four objectives.

4. Identify the five control logic structures.

5. Analyze a given problem and use the correct format and technique in writing the solution to that problem in terms of a flowchart.

INFORMATION

PROBLEM SOLVING

This section will concentrate on problem-solving methods and program design. Remember, a correct program design is really an ordered list of all processes you would have to perform if you were the computer. If a design does not contain enough information for you to solve the problem, it is certainly not detailed enough for you to tell an "idiot with the speed of light" (the computer) how to solve the problem.

Problem Solution Methods

Selecting and analyzing problem solution methods is important throughout the problem definition phase of computer programming. Although a person may solve different problems in what appears to him to be the same way, he in fact gets to the solution in one of (or a combination of) the following four methods:

1. Direct.

2. Enumerating.

3. Scientific trial and error.

4. Simulation.

The first thing to be done in all of these methods is to put the problem into a proper framework. This framework consists of three parts:

1. Identify the known parts and their relationship to the solution.

2. Identify the solution.

3. Identify the unknown or variable parts.

Having done this, it becomes a matter of building a string of relationships between the known parts of the problem and the solution to the problem. Unfortunately, it is difficult to know which relationships will lead to the most direct solution to the problem.

If a person moves efficiently from known parts to solutions, he probably has more "feel" for the problem than someone who stumbles about before finding the right path.

DIRECT METHOD. The direct method is probably the most widely used of all problem-solving methods and, therefore, it is so ingrained and habitual that it appears to be purely intuitive. For this reason, it is extremely difficult to explain or to analyze it scientifically. If you consider being thirsty a problem, then the solution to the problem would be "go to the water fountain, and get a drink of water." This would be an example of a direct solution method.

ENUMERATING METHOD. This method is also widely used and is so simple it might seem that there is little reason for even listing it. It consists of checking every possible entity that could solve the problem. In other words, if one wished to find the heaviest book in a library, he would weigh each one, check each weight, and select the book with the largest weight.

There are two requirements for using this method:

1. All of the possible solutions must be known.

2. There must be some criterion against which to match the possibilities.

SCIENTIFIC TRIAL AND ERROR METHOD. Similar to the enumerating method, scientific trial and error is used where the number of possible solutions is very large. A guess is made at the answer, attempting to come as close as possible to the solution. Then the guess is examined for what it has done for the problem. Based on the results of this examination another guess is made, hopefully even closer to the solution than the previous guess. These steps are repeated until the solution is found. This may at first appear to be a very primitive approach, but it is very versatile and useful. It is also well suited to computers, as it is basically an iterative process method.

An excellent use of Scientific Trial and Error would be a common method for taking the square root of a number, using a basic four function calculator. Call "X" the number we want to take the square root of. Then let "G" (the initial guess) be equal to X divided by 2 (i.e., $G = X/2$). Now, using algebraic notation:

$$SQR = [(X/G) + G]/2$$

If SQR and G are equal (or very close) stop calculating; otherwise, let G equal SQR and reevaluate the equation.

SIMULATION METHOD. The simulation method of problem solving goes a step beyond the mathematical model of an algorithm. This method attempts to make an actual working model of the real world. The working model is set in motion to see what possible outcomes might actually occur for a given set of data.

For example, assume the town of Loannisport has a traffic problem. Every day at 5:30 P.M. traffic seems to be tied up at the main intersection of the town for more than an hour. The town council has decided that a more modern traffic signal will get things moving more quickly. Therefore, the problem to be resolved is the timing and display sequence of the light. The simulation method seems to be in order here.

Data on traffic flow is gathered, along with other data that may influence the problem, such as the number of pedestrians using the intersection, schools in the area, maximum speed limits consistent with safety, etc. A mathematical algorithm is built with this data and a variable. The variable is the various timing and display sequences of the traffic light. The solution to the problem is a timing and display sequence which causes the least congestion at the intersection. The algorithm is written for the computer, and the program is run using different values which yield various answers. An answer and a solution are two different entities.

Note that an answer relates to a set of problem specifications where a variable has been given a specific value, and that value is used in computation. Every time a different sequence for the light was used in the mathematical algorithm, the computer came up with a different answer. Only one, or at most a few answers, can be a satisfactory problem solution.

The solution to the traffic problem at Loannisport was the installation of a properly sequenced light at the main intersection. The answer (satisfactory problem solution) was the specific light sequence which produced a minimum amount of congestion at the intersection.

The Algorithmic Statement of Problem Solutions

An algorithm is the set specifications or instructions for solving a problem. A mathematical algorithm might be expressed as follows: take a number, multiply it by itself, add to it twice itself, subtract one, and call this the result. In algebra this algorithm can be stated:

$$R = X^2 + 2X - 1$$

Note that an algorithm need not be mathematical.

A good algorithm has two properties. First it must be clear. Each step must have one--and only one--interpretation. Everyone reading the step must be able to accomplish it in the same way, arriving at the same result. Second, the algorithm must stop.

To demonstrate the importance of these two properties, consider the fifth step of an algorithm for making a good cup of coffee, written: "Add 1 teaspoon of sugar until the coffee is sweet enough." Note that the concept of sweetness may vary from person to person, is therefore unclear, and violates property one. Also, a person with a deformed sense of taste may never stop adding sugar to the cup, which violates property two. A better fifth step would be, "Add one teaspoon of sugar," followed by step six, "Taste the coffee," step seven, "If it is not sweet enough, add another teaspoon of sugar," and step eight, "Repeat steps six and seven until the coffee is sweet enough or ten teaspoons of sugar have been added." This modified algorithm is clearly stated and will stop.

For the majority of your work we are interested in algorithms that solve the problem. However, a word of caution is in order. Because your algorithm solves the problem does not mean it is a good solution to the problem. An algorithm must only be clear and terminate. For example, an algorithm to empty the water in a pond might be:

1. "Remove the water from a pond with an 8-ounce glass."

2. "Stop when the pond is empty."

This is a straight-forward unambiguous task that will eventually stop, but it is not a good solution to the problem.

DIRECT ALGORITHM. A direct algorithm is one that is made up of a number of known steps, and a result is determined by these steps. The example $R = X^2 + 2X - 1$ is a direct algorithm. You can tell just by looking at the algebraic notation just what and how much is involved in obtaining a result.

REPETITIVE ALGORITHM. A repetitive (or iterative) algorithm is one in which some or all of the steps of the process are repeated. An example of an iterative algorithm was given in the example about making a cup of coffee. Step six and step seven were repeated until the coffee was sweet enough or ten teaspoons of sugar had been added.

INDIRECT ALGORITHM. Very simple, an indirect algorithm is one that is not direct. Remember that a direct algorithm has a number of known steps, and indicates how much work must be done by looking at it. Consider the previous example which specified the removal of all water from a pond with an 8-ounce glass. It (like the coffee algorithm) is iterative, but how many times the process of dumping one glass of water must be repeated is unknown at the beginning.

Problem Definition

Quite often when a problem is defined in detail, the problem solution becomes evident. Ask yourself some questions about this statement, such as: What do we mean by define in detail? How can we define a complex problem in detail? Or, even, why not just do the job and define the problem as we go? These are all valid questions that will be answered as we progress through this section.

First, why even worry about defining the problem? To illustrate the need for a complete problem definition, suppose you were given some Air Force orders sending you on temporary duty. The orders stated only:

Report to Colonel Arnold Flakbait
123 S. Main St, Rm 461
Washington

at 0800 hours 3 days from today.

Do you have the information you will need to perform the job you have been given? NO? You're right. How are you going to get there? Plane? Car? Train? Stagecoach? What do we mean by today? Now? The day the orders were printed? The day Colonel Flakbait said "Send G.I. Joe to Washington"? For that matter, Washington where? D.C.? State? Arkansas? Now, suppose you get the dates straightened out, discover you are to drive, and Washington is in the Northeast U.S. What do you do? Sit down and decide which highways you have to take to get to Washington, D.C.? I hope you don't go to D. C. because Colonel Flakbait is expecting to meet you in Washington, Connecticut.

What? You say the Air Force would not issue any orders like the one above? True! However, you may rest assured that as long as you are a computer programmer, people will bring you problems that are not even as well defined as our proposed TDY orders.

Consider, for example, the businessman who wanted to write a program to computerize his stock inventory. His program consisted of:

"Dear Computer,

Please take the following information and save it so that you can give it back to me in the way I want it when I want it."

Our businessman followed this statement with his store's inventory list. He was quite serious and felt he had given the computer all the information it needed to do what he wanted done. Do you know what he wanted?

Before we jump into writing a program to solve a given problem, there are a number of things that must be done. One of the things to be done at the beginning is to eliminate as many assumptions as possible. Assumptions will probably get you into more trouble than anything else. Do not assume that you know what the problem is on the first telling or reading. Unfortunately, the language used to relate the problem to the programmer is usually English. English can have infinite shades of meaning to different people, which is bad enough, but add a smattering of technical jargon, and who can be sure what is meant. Remember the note from the boss to his employees? "I'm sure that you think you understand what you thought I said, but what I meant to say is not what I think you have assumed I meant."

Now that you are well aware of the pitfalls that can be found in the realm of problem solving, here are some rules that will help you steer around these pitfalls.

The first rule is to write everything down! Don't rely on your memory because little things will slip by you, or be forgotten. If you don't believe me, what room number is Colonel Flakbait going to be in? No fair peeking! The reason you had trouble remembering the room number is because it didn't seem important when you read it. Very often, facts that seem insignificant or unimportant when the problem is first presented to you can turn out to be the vital key to the successful solution of the problem.

The second rule is to solve only simple problems. But you know that people are not going to bring you simple problems to solve. They will solve those themselves. You are going to be given some problems that will put you into the mumble mode. You must somehow make the complex problem simple, which is easier said than done. But if you were to examine a complex problem closely, you would discover that it is made up of smaller sub-problems--each of which is simpler than the whole problem by itself. Each sub-problem only contains a part of the problem, much like breaking a pencil in half. Each piece is smaller than the whole, but together they make up the whole.

THE IDEA OF STRUCTURED PROGRAMMING

It has been discovered recently that computer programs can be written with a high degree of structure, which permits them to be more easily understood for testing, maintenance, and modification. With Structured Programming, control branching is entirely standardized so that code can be read from top to bottom, without having to trace the branching logic as is typical for code generated in the past. Structured Programming represents a new technical standard which permits better enforcement of design quality for programs. It corresponds to principles in hardware design, where it is known that all possible logic circuits can be formed out of a small collection--AND, OR, NOT--of standard component circuits.

In Structured Programming, programmers must think deeper, but the end result is easier to read, understand, and maintain. The standards of Structured Programming are based on new mathematical theorems and do not require case-by-case justification. Just as it is the burden of a professional engineer to be able to design logic circuits out of certain basic components, so it is the burden of a professional programmer to write programs in a structured way, using only recently standardized branching conventions.

Top Down Programming

Structured Programming also enhances the development of programs in a "top down" form, in which major programs can be broken into smaller programs through a combination of code and the designation of dummy programs called program stubs, which are referenced or called by that code. By writing the code which calls the program stubs before the stubs themselves are developed, the interfaces between the calling and the called programs are defined completely so that no interface problems will be encountered later.

The result of the systematic, disciplined approach of Structured Programming is higher precision programming than was accomplished before. The testing of such programs is accomplished more rapidly, and the final results are programs which can be read, maintained, and modified by other programmers with much greater facility.

Structured Programming Theory

Any program, no matter how large or complex, can be represented as a set of flowcharts. Structured Programming theory deals with converting large and complex flowcharts into standard forms so that they can be represented by iterating and nesting a small number of basic and standard control logic structures. A sufficient set of basic control logic structures consists of three members (see figure 4-1):

1. A sequence of two or more operations (SEQUENCE).

2. A branch to one operation if the condition is True and a branch to another operation if the condition is False (IFTHENELSE).

3. Perform some operation while some condition is true (DOWHILE).

The basic structure theorem, due in original form to Bohm and Jacopini*, is that any flowchart can be represented in an equivalent form as an iterated and nested structure in these three basic and standard figures.

Note that each structure has only one input and one output, and can be substituted for any box in a structure, so that complex flowcharts can result. The key point is that an arbitrary flowchart has an equivalent representative in the class so built up.

The structure theorem demonstrates that programs can be written in terms of IFTHENELSE and DOWHILE statements. The idea of an unconditional branch and corresponding statement label is never introduced in these basic structures, and is thus never required in a representation.

There is no compelling reason in programming to use such a minimal set of basic figures, and it appears practical to augment the basic set with two variations in order to provide more flexibility. The variations are: (See figure 4-1.)

1. Perform some operation until some condition is False (DOUNTIL).

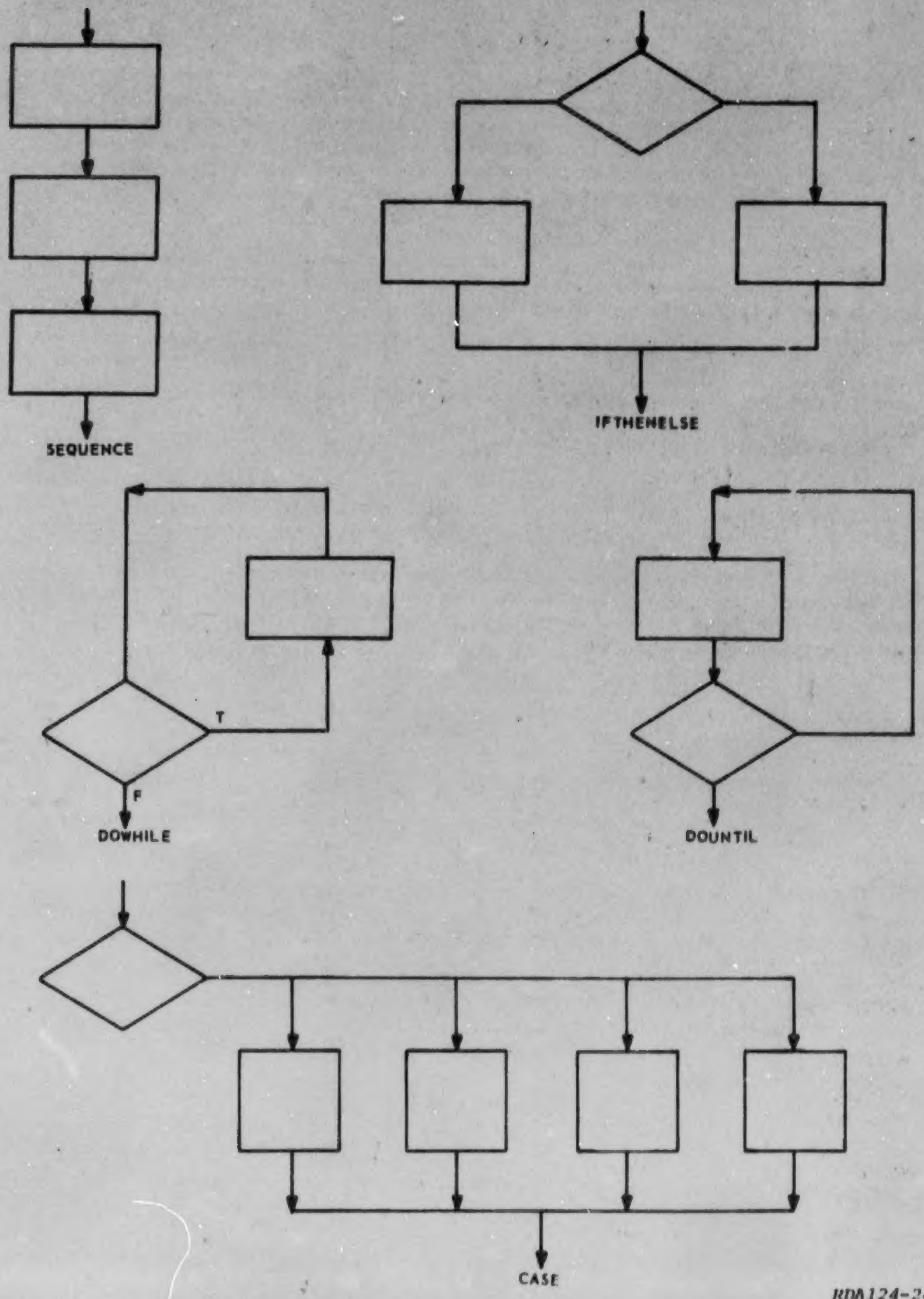
2. Branch to more than two operations depending on the value of some condition (CASE).

DOWHILE provides an alternative form of looping structure, while CASE is a multi-branch, multi-join control structure in which it is convenient to express the processing of one of many possible unique occurrences.

A major characteristic of programs written in these structures is that they may be literally read from top to bottom; there is never any "jumping around" as is so typical in trying to read code which contains unconditional branches. This property of readability is a major advantage in developing, testing, maintaining, or otherwise referencing code at later times.

---

*Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the Association for Computing Machinery, Volume 9, No. 4, May 1966.



RDA124-24

Figure 4-1. Controlled Logic Figures

Another advantage, of possibly even greater benefit, is the additional program design work that is required to produce such structured code. The programmer must think through the processing problem, not only writing down everything that needs to be done, but writing it down in such a way that there are no afterthoughts with subsequent jump-outs and jump-backs, no indiscriminate use of a section of code from several locations because it "just happens" to do something at the time of the coding. Instead, the programmer must think through the control logic of the module complete at one time in order to provide the proper structural framework for the control. This means that programs will be written in a much more uniform way because there is less freedom for arbitrary variety.

Such a program is much easier to understand than an unstructured logical jumble; readability has been improved. Because of its simplicity and clear logic, it minimizes the danger of the programmer's overlooking logical errors during implementation; reliability has been improved. And improved readability, in combination with the great simplicity obtained by Structured Programming, naturally leads to improved maintainability. Further, because structured code is simple, a programmer can control and understand a much larger amount of code. With increased productivity, programming costs can be reduced.

In conclusion, Structured Programming may be defined as an application of standards and/or guidelines to computer programming with certain objectives in mind. The objectives are:

1. Readable code.
2. Reliable code.
3. Maintainable code.
4. Increased programmer productivity.

Table 4-1 shows common symbols and their descriptions.

Table 4-1

COMMON SYMBOLS AND THEIR DESCRIPTIONS

Function	Symbol	Description
Assignment	=	Set the left item equal to the contents (value) of the right expression.
Addition	+	Add the values or contents of the two adjacent terms.
Subtraction	-	Subtract the value of the right term from the value of the left term.
Multiplication	*	Multiply the values of the two adjacent terms.
Division	/	Divide the value of the left term by the value of the right term.
Exponentiation	**	Raise the value of the left term to the power indicated by the value of the right term.

FLOWCHARTING

The flowchart problems in this text are designed to teach students flowcharting techniques from a general problem-solving viewpoint, not from a specific compiler language viewpoint. With a good foundation in flowcharting techniques, the student should be able to readily apply them to the language he is using (i.e., FORTRAN, GMAP, or COBOL).

The following conventions will apply to the flowchart examples in this text:

Some problems may reference files in memory, which for the purposes of this text will be treated as tables in core memory (similar to the way COBOL treats tabular data). The records of these files (and the fields within these records) may be accessed by referencing the file's symbolic name (or the field's symbolic name) with an integer constant or variable subscript. For example, consider the file named STUFIL with record format:

NAME	ADDR	COURSE	GD
1 - 20	21 - 45	46 - 69	70

Record five of STUFIL could be accessed by writing STUFIL(5). Likewise, the NAME field of record 44 could be accessed by writing NAME(44), the COURSE field of record 15 could be accessed by writing COURSE(15), or the GD field of record N could be accessed by writing GD(N), where N equals an integer constant. Remember this method of accessing records and fields within records is for problems in this text, and may not work in some compiler languages (i.e., FORTRAN).

Refer to Appendix A for a description of standard flowchart symbols.

Sequence Flowchart

A sequence flowchart is simply a logical sequence of operations to be performed exactly one time.

Problem 1

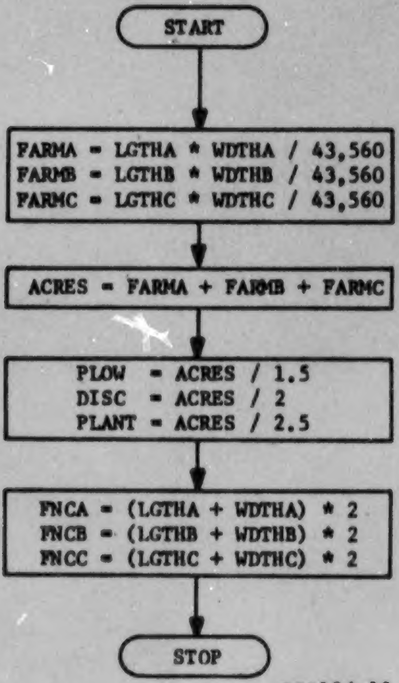
A farmer has three farms (FARMA, FARMB, and FARMC). The dimensions (in feet) of these farms are called LGTHA, WDTHA, LGTHB, WDTHB, LGTHC, and WDTHC. Find how long it will take to plow, disc, and plant all three farms.

The farmer can plow 1.5 acres an hour, disc 2 acres an hour, and plant 2.5 acres an hour. Also, find how many feet of fencing it will take to fence each farm and how many acres are in each farm. The following variables will be used to represent the answers: PLOW - the time needed to plow all three farms; DISC - the time needed to disc all three farms; PLANT - the time needed to plant all three farms; FNCA - the amount of fencing needed for FARMA; FNCB - the amount of fencing needed for FARMB; FNCC - the amount of fencing needed for FARMC; FARMA - the number of acres in FARMA; FARMB - the number of acres in FARMB; FARMC - the number of acres in FARMC; and ACRES - the number of acres in all three farms.

Solution 1

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
FNCA = (LGTHA + WDTHA) * 2	8
FNCB = (LGTHB + WDTHB) * 2	9
FNCC = (LGTHC + WDTHC) * 2	10
FARMA = LGTHA * WDTHA / 43,560	1
FARMB = LGTHB * WDTHB / 43,560	2
FARMC = LGTHC * WDTHC / 43,560	3
ACRES = FARMA + FARMB + FARMC	4
PLOW = ACRES / 1.5	5
DISC = ACRES / 2	6
PLANT = ACRES / 2.5	7

STEP THREE (Flowchart the sequence of operations) (See figure 4-2.)


START
FARMA = LGTHA * WDTHA / 43,560
FARMB = LGTHB * WDTHB / 43,560
FARMC = LGTHC * WDTHC / 43,560
ACRES = FARMA + FARMB + FARMC
PLOW = ACRES / 1.5
DISC = ACRES / 2
PLANT = ACRES / 2.5
FNCA = (LGTHA + WDTHA) * 2
FNCB = (LGTHB + WDTHB) * 2
FNCC = (LGTHC + WDTHC) * 2
STOP

RDA124-19

Figure 4-2. Sequence Flowchart

Branching Flowchart

A branching flowchart is simply a sequence flowchart which allows one or more of its operations to be decisions, providing a break in the sequence of operations performed.

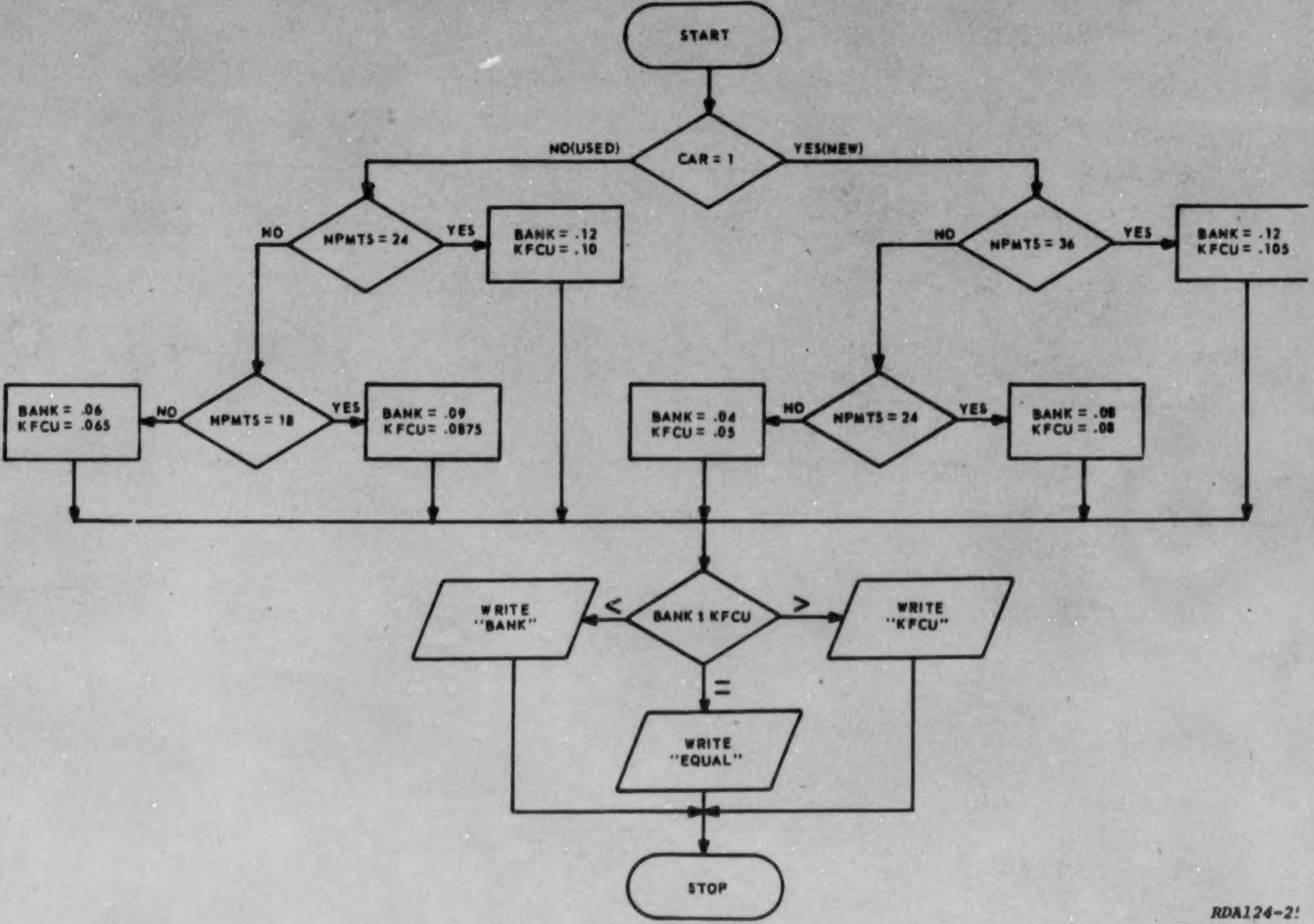
Problem 2

If CAR contains the status of an auto (new or used) and NPMTS contains the number of months the auto is to be financed, then set KFCU and BANK to the appropriate interest rate based on Chart 4-1 and compare them to determine where to finance the auto (for the lowest interest rate). The following variables are given: CAR - status of the auto (1 = New or 2 = Used); NPMTS - the number of months auto is to be financed; KFCU - the interest rate the credit union will finance an auto; and BANK - the interest rate the bank will finance an auto.

Solution 2

STEP ONE (list the operations to be performed)	STEP TWO (number the sequence of operations)
Is auto new (CAR = 1)?	1
Yes, go to 2.	
No, go to 4.	
Is new auto NPMTS = 36?	2
Yes, go to 6.	
No, go to 3.	
Is new auto NPMTS = 24?	3
Yes, go to 8.	
No, go to 10.	
(Since NPMTS $\neq$ 36 or 24, then NPMTS = 12)	
Is used auto NPMTS = 24?	4
Yes, go to 12.	
No, go to 5.	
Is used auto NPMTS = 18?	5
Yes, go to 14.	
No, go to 16.	
(Since NPMTS $\neq$ 24 or 18, then NPMTS = 12)	
Compare BANK to KFCU	18
GT, go to 20.	
EQ, go to 21.	
LT, go to 19.	
BANK = 12.0	6
KFCU = 10.5	7
Go to 18.	
BANK = 8.0	8
KFCU = 8.0	9
Go to 18.	
BANK = 4.0	10
KFCU = 5.0	11
Go to 18.	
BANK = 12.0	12
KFCU = 10.0	13
Go to 18.	
BANK = 9.0	14
KFCU = 8.75	15
Go to 18.	
BANK = 6.0	16
KFCU = 6.5	17
Go to 18.	
Write "BANK"	19
Write "KFCU"	20
Write "EQUAL"	21
Stop processing	22

STEP THREE (Flowchart the sequence of operations.) (See figure 4-3.)



RDA124-2

Figure 4-3. Branching Flowchart (Multiple Decision)

152

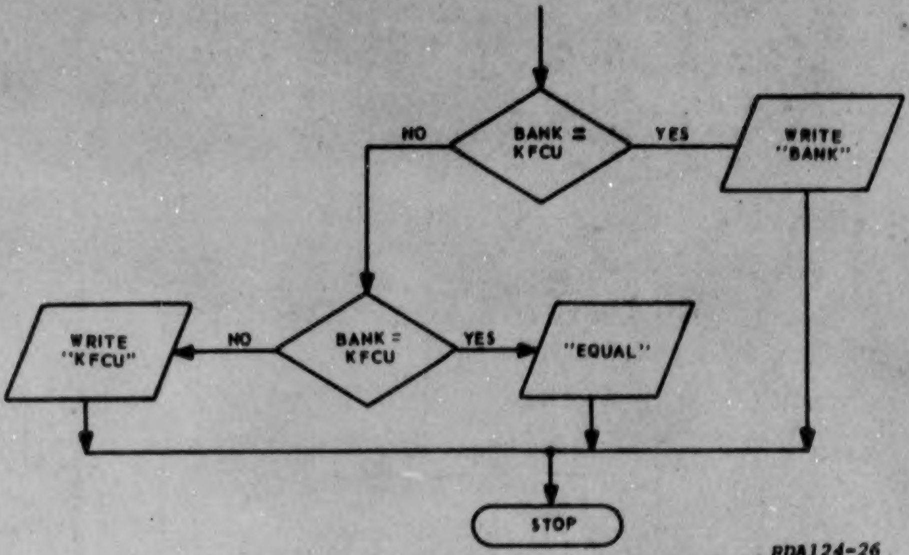


Figure 4-4. Alternate Compare Operation

Loop Flowchart

A loop flowchart is a branching flowchart with the addition of a loop. The four parts of a loop are:

1. Initialize - set the counter to zero (performed outside the loop).

2. Test - make a decision to see if the number in the counter is equal to the total number of times the operation is to be performed.

3. Perform - this includes the part of the flowchart that shows the operations to be performed.

4. Modify - change the counter to show the number of times the operation has been performed.

A variation on the above routine is to initialize the counter to the number of times the operation is to be performed, test for zero, and modify the counter by decrementing the counter by one each time the operation is performed.

Problem 3

Continue with the problem solved in the Branching Flowchart section. This process, depicted in figure 4-3 (refer to this predefined process as the symbolic name FINANCE), determines where one car can be financed economically. Show how this process might be used to process a file which contains purchase information on 100 automobiles. Refer to Problem 2 for the variables used in this problem.

147

Solution 3

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
COUNT = 0	1
Is COUNT = 100? Yes, go to 7. No, go to 3.	2
Read card from file	3
Call predefined process FINANCE	4
COUNT = COUNT + 1	5
Go to 2.	6
Stop processing	7

STEP THREE (Flowchart the sequence of operations.) (See figure 4-5.)

Search Flowchart

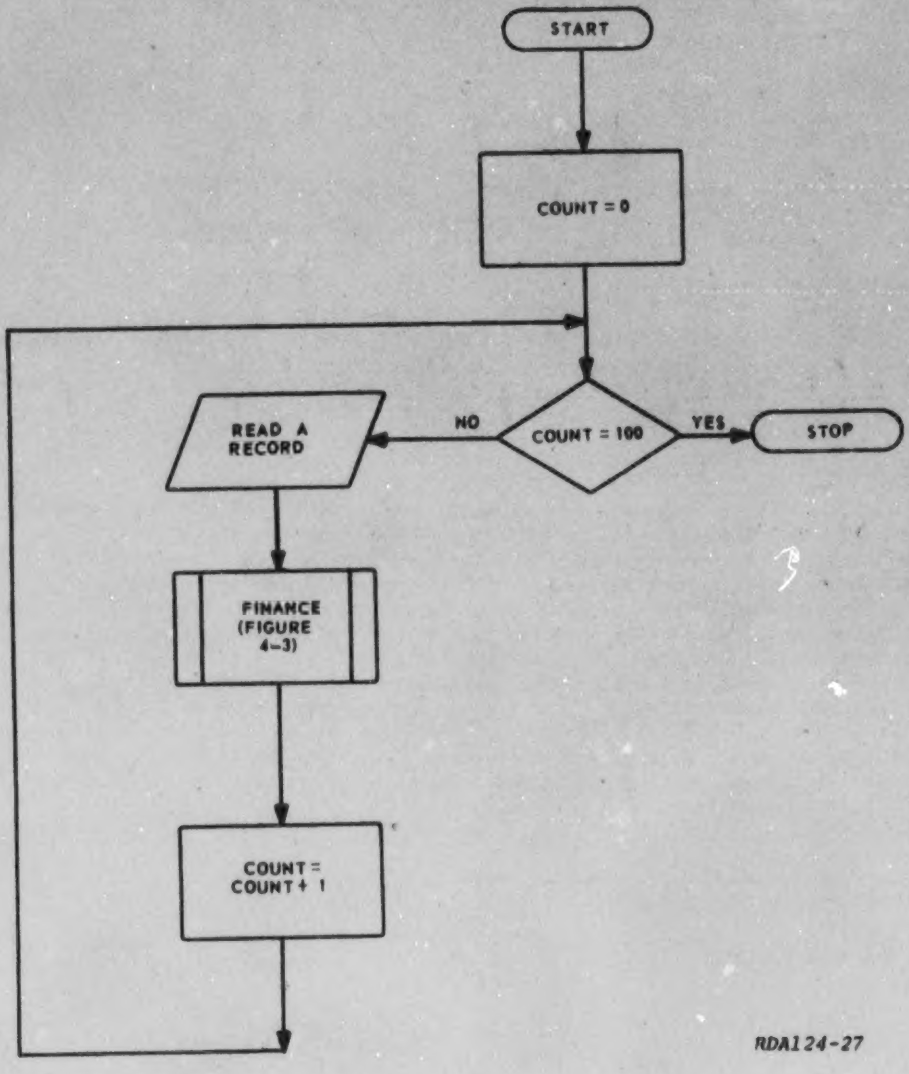
A search flowchart is one which looks for a specific record within a file, based on some key field. The method used to search for a record within a file is basic to all other manipulations of records within the file, since you must first locate the desired record before you can change or delete it.

Problem 4

A large car rental company has data pertaining to its cars in a computer file. CAR is a file in memory with up to 500 records, each containing the following variables: SERN - car serial number; MLGE - car mileage; MNDS - mileage next service is due; and VRM - vehicle retirement mileage. ENDD is another variable which indicates the number of records in CAR. There is a stack of 50 cards in the card reader, each containing the following variables: SERNO - car serial number and PMLGE - present mileage of car. For each card read, use SERNO to locate the record in CAR with the same serial number. Replace the old mileage with the new mileage figure (PMLGE) and check to see if the mileage has reached or exceeded the vehicle retirement mileage or the next service due mileage. If so, print out "RETIRE" or "SERVICE" as applicable, and also print the serial number, mileage, mileage next service due, and vehicle retirement mileage. If the serial number from any card read does not have a matching serial number in CAR, print an error message.

SERN	MLGE	MNSD	VRM
1-8	9-14	15-20	21-26

SERNO	PMLGE
1-8	9-14



RDA124-27

Figure 4-5. Loop Flowchart

Solution 4

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
Read a card	3
K = 1	2
Is card reader empty? Yes, go to 17. No, go to 2.	1
Is SERNO = SERN(K)? Yes, go to 9. No, go to 5.	4
MLGE(K) = PMLGE	9
K = K + 1	5
Is K GE ENDD? Yes, go to 7. No, go to 4.	6
Write error message.	7
Go to 1.	8
Is MLGE(K) GE VRM(K)? Yes, go to 12. No, go to 11.	10
Is MLGE(K) GE MNSD(K)? Yes, go to 13. No, go to 1.	11
Write "RETIRE"	12
Go to 15.	13
Write "SERVICE"	14
Write SERN(K), MLGE(K), MNSD(K), VRM(K)	15
Go to 1.	16
Stop processing	17

STEP THREE (Flowchart the sequence of operations.)

See figure 4-6.

156

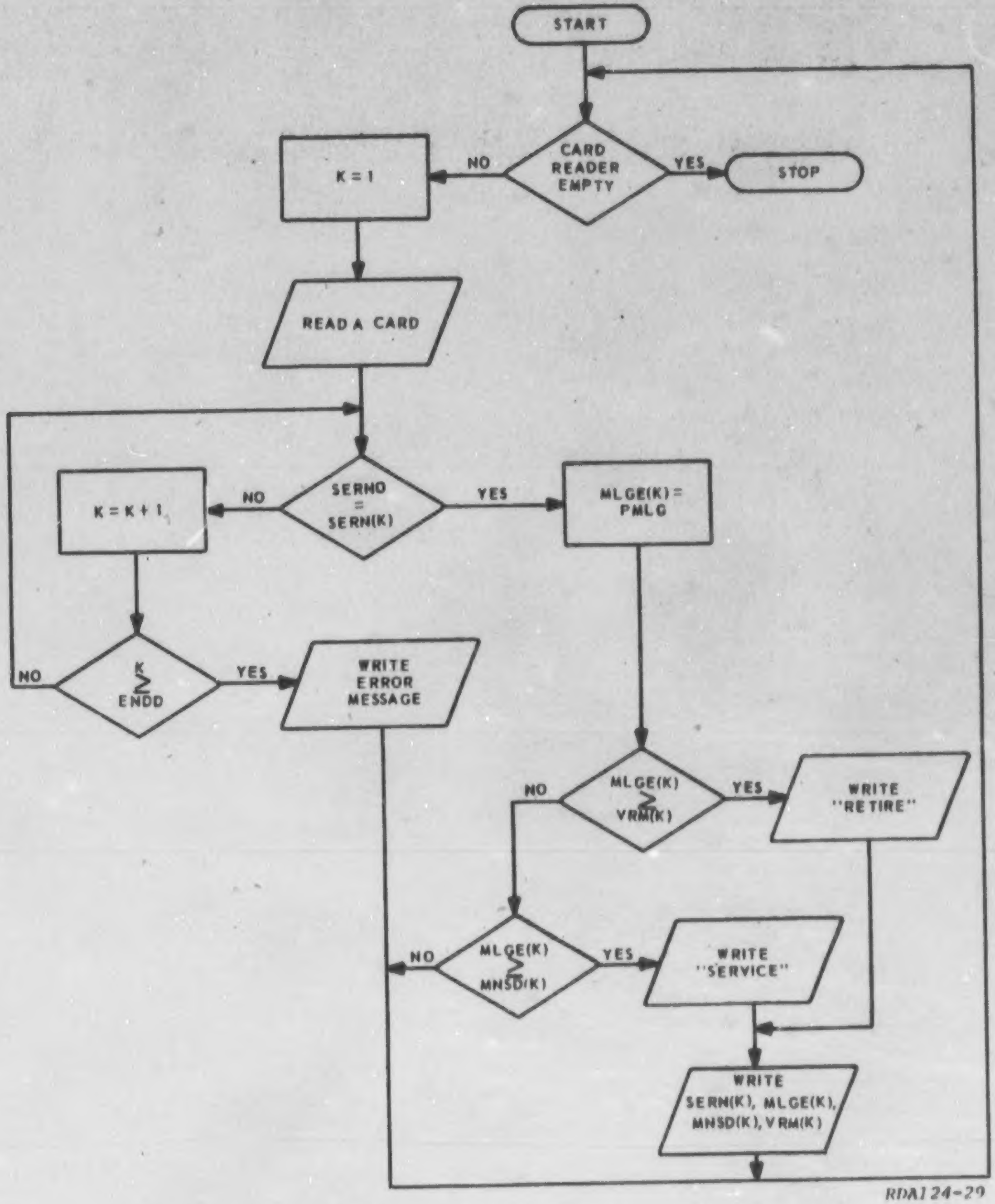


Figure 4-6. Search Flowchart

Sort Flowchart

A sort flowchart is actually a modified search flowchart which arranges file records in either ascending or descending order. One or more key fields within a record are used to sort a file. The sort process utilizes a key field to search through a file, comparing the key field of one record with the key field of the next sequential record until two records are found to be out of order. These records must then swap places in the file. Next, go back to the beginning of the file and repeat this process of searching and swapping until you reach an end of file. This process is a Simple Exchange Sort. Variations of this sort and alternate sort routines may be presented by your instructor in this block.

Problem 5

STU is a file in memory with up to 100 records, each containing the following variables: NAME - name of student and PCG - percentage grade of student. NENT is another variable which indicates the number of records in STU. STU is sorted in alphabetical order. Sort the file in descending order, based on key field PCG. BUFF is a temporary storage location.

NAME	PCG
1-17	18-20

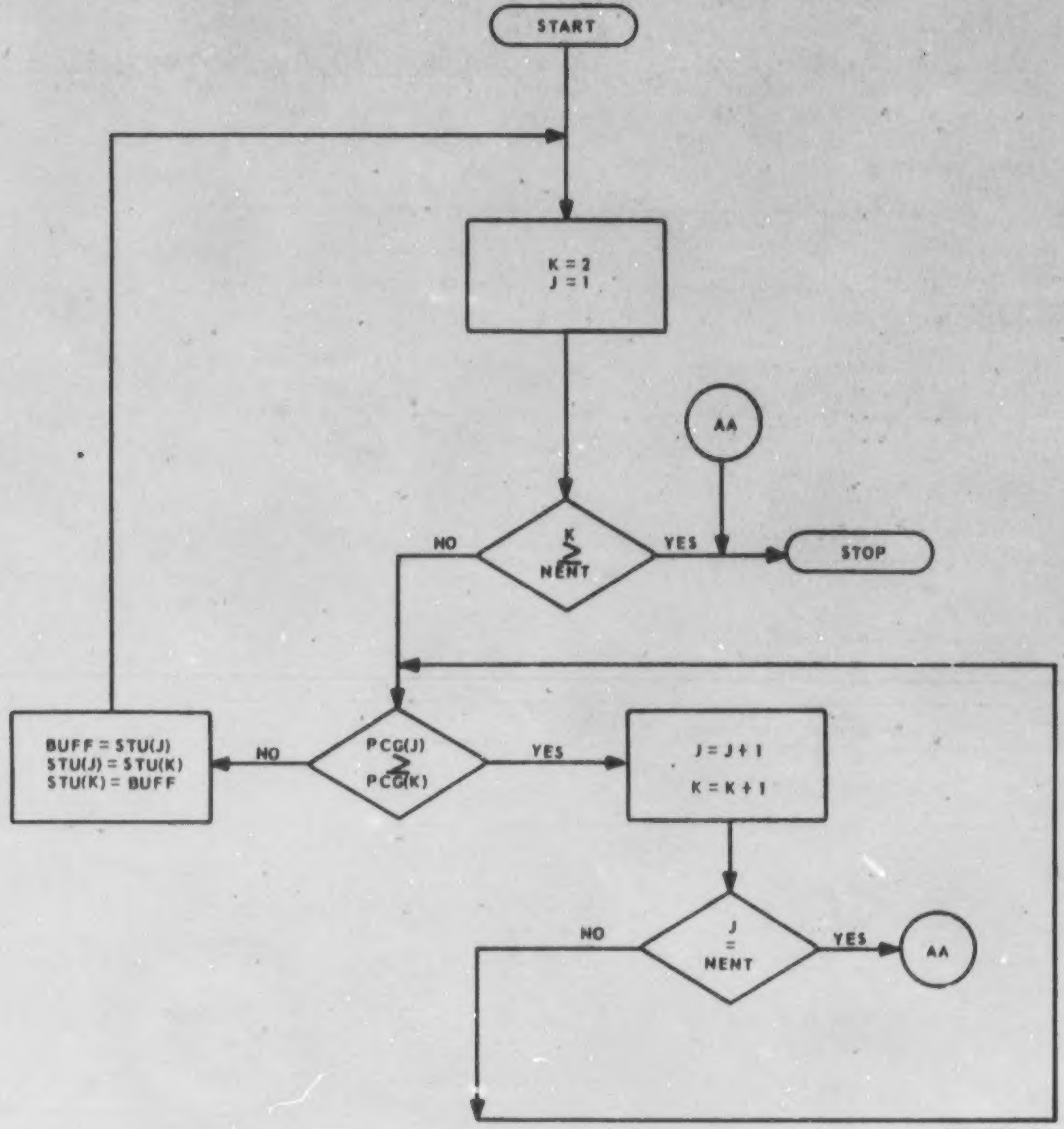
BUFF
1-20

Solution 5

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
J = 1	1
K = 2	2
Is K GT NENT? Yes, go to 10. No, go to 1.	3
Swap STU(J) with STU(K).	8
Go to 1.	9
Is PCG(J) GE PCG(K)? Yes, go to 5. No, go to 8.	4
J = J + 1	5
K = K + 1	6
Is J = NENT? Yes, go to 10. No, go to 4.	7
Stop processing	10

STEP THREE (Flowchart the sequence of operations)

See figure 4-7.



RDA124-28

Figure 4-7. Sort Flowchart

Insertion Flowchart

An insertion flowchart is also a modified search flowchart, which inserts one or more records into a file at a specified location. If the file is in random order, records are inserted at the end of the file (if there is room). If the file is in ascending or descending order, records are inserted at the proper location within the file (if there is room). This type of insertion consists of the basic search to find the proper location for the new record, exchanging the new record for the old, moving the old record to the next location, and moving that record to the next location, and so on until the last meaningful location in the file is reached. After inserting the last record, a counter must be incremented to reflect the new file entry. This completes the process for single insertion. Repeat the process for multiple insertion of records.

Records can be inserted only into a variable-length file, not into a fixed-length file. When it is necessary to add records to a fixed-length file, a new file must be built of sufficient size to hold all old and new records.

Problem 6

AUTO is a file in memory with up to 300 records, each containing the following variables: SERN - car serial number; MLGE - present car mileage; CST - car cost; and GAS - miles per gallon rating. LAST is another variable which indicates the current number of records in AUTO. AUTO is arranged in random order. Insert a single record NEW into AUTO (if there is room).

SERN	MLGE	CST	GAS
1-6	7-12	13-18	19-20

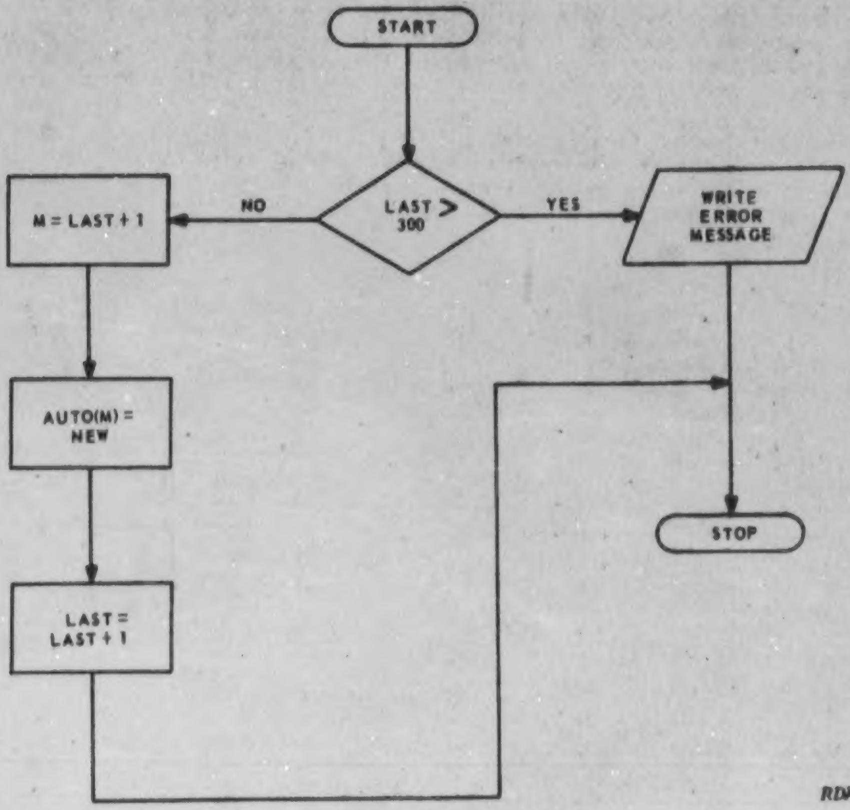
NSERN	NMLGE	NCST	NGAS
1-6	7-12	13-18	19-20

Solution 6

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
Is LAST GT 300? Yes, go to 2. No, go to 4.	1
Write error message.	2
Go to 7.	3
M = LAST + 1	4
AUTO(M) = NEW	5
LAST = LAST + 1	6
Stop processing.	7

STEP THREE (Flowchart the sequence of operations.)

See figure 4-8.



RDA124-30

Figure 4-8. Insertion Flowchart
(Random Single Insertion)

Problem 7

NYPD is a file in memory with up to 2,000 records, each containing the following variables: NAME - employee name and SSN - employee social security number. ENDD is another variable which indicates the number of records in NYPD. NEW is a file in memory with up to 50 records, each containing the following variables: KNAME - new employee and KSSN - new employee social security number. STOPP is another variable which indicates the number of records in NEW. Both files are in descending order, based on social security number. Insert all records of NEW into the proper location within NYPD (if there is room).

NAME 1-20	SSN 21-29

KNAME 1-20	KSSN 21-29

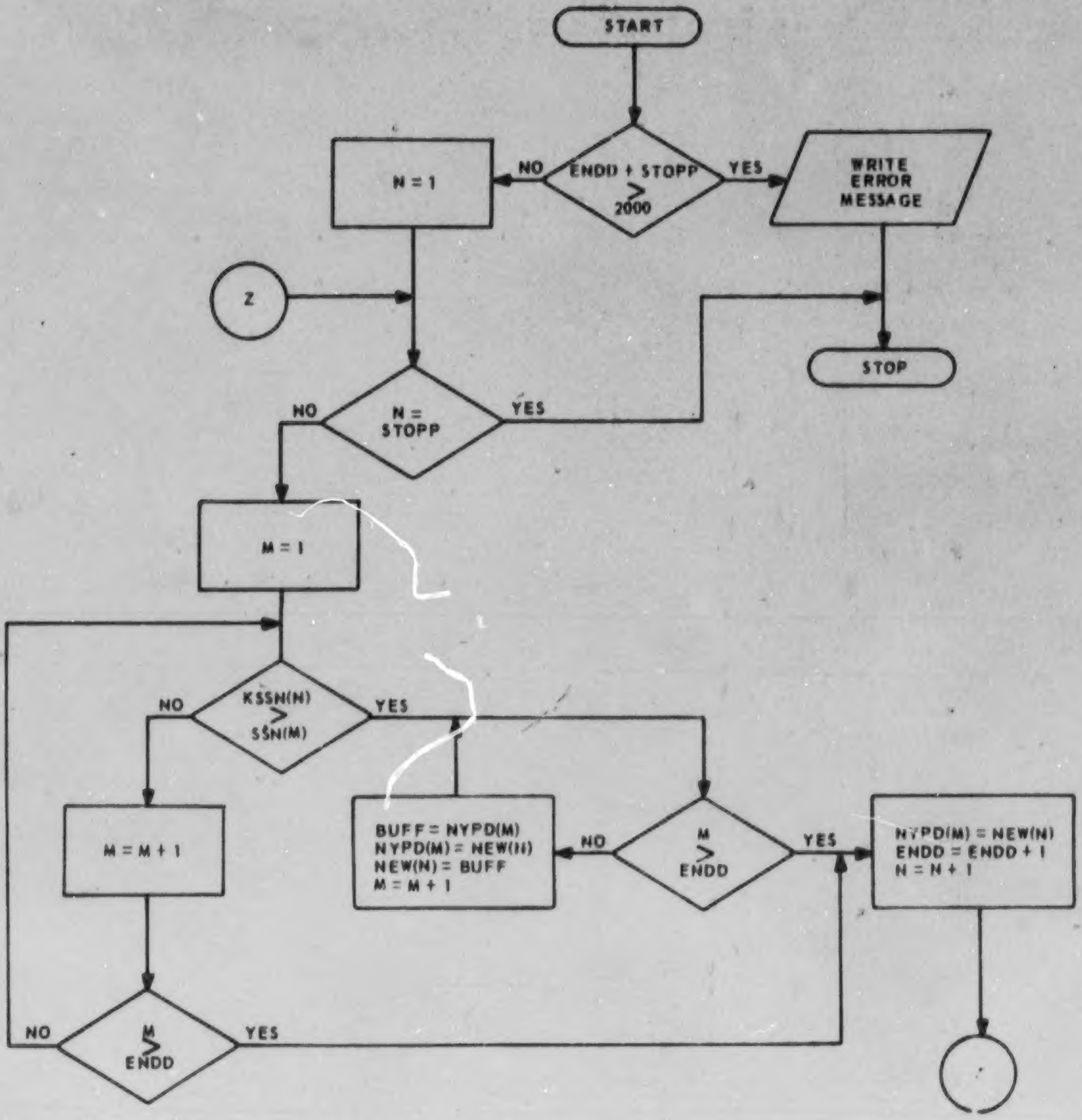
Solution 7

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
Is ENDD + STOPP GT 2,000? Yes, go to 4. No, go to 2.	1
Write error message.	4
Go to 20.	5
N = 1	2
Is N GT STOPP? Yes, go to 20. No, go to 6.	3
M = 1	6
Is KSSN(N) GT SSN(M)? Yes, go to 10. No, go to 8.	7
M = M + 1	8
Is M GT ENDD? Yes, go to 16. No, go to 7.	9
Is M GT ENDD? Yes, go to 16. No, go to 11.	10
NYPD(M) = NEW(N)	16
ENDD = ENDD + 1	17
N = N + 1	18
Go to 3.	19
BUFF = NYPD(M)	11
NYPD(M) = NEW(N)	12
NEW(N) = BUFF	13
M = M + 1	14
Go to 10.	15
Stop processing.	20

STEP THREE (Flowchart the sequence of operations.)

See figure 4-9.

162



FDA124-31

Figure 4-9. Insertion Flowchart (Ordered Multiple Insertion)

157

Deletion Flowchart

A deletion flowchart is also a modified search flowchart, which deletes one or more records from a file at a specified location. Deletion of a single record consists of a basic search to find the record to be deleted, decrementing the file counter to reflect the deletion, and packing the file to eliminate empty locations caused by the deletion. Repeat this process for multiple deletions. Records may be deleted from both fixed-length and variable-length files.

Problem 8

NYPD is a file in memory with up to 2,000 records, each containing the following variables: NAME - employee name and SSN - employee social security number. ENDD is another variable which indicates the number of records in NYPD. GONE is a file in memory with up to 50 records, each containing the following variables: KNAME - retired employee and KSSN - retired employee social security number. STOPP is another variable which indicates the number of records in GONE. Both files are in ascending order, based on social security number. Delete all records for retired employees from NYPD.

NAME 1-20	SSN 21-29

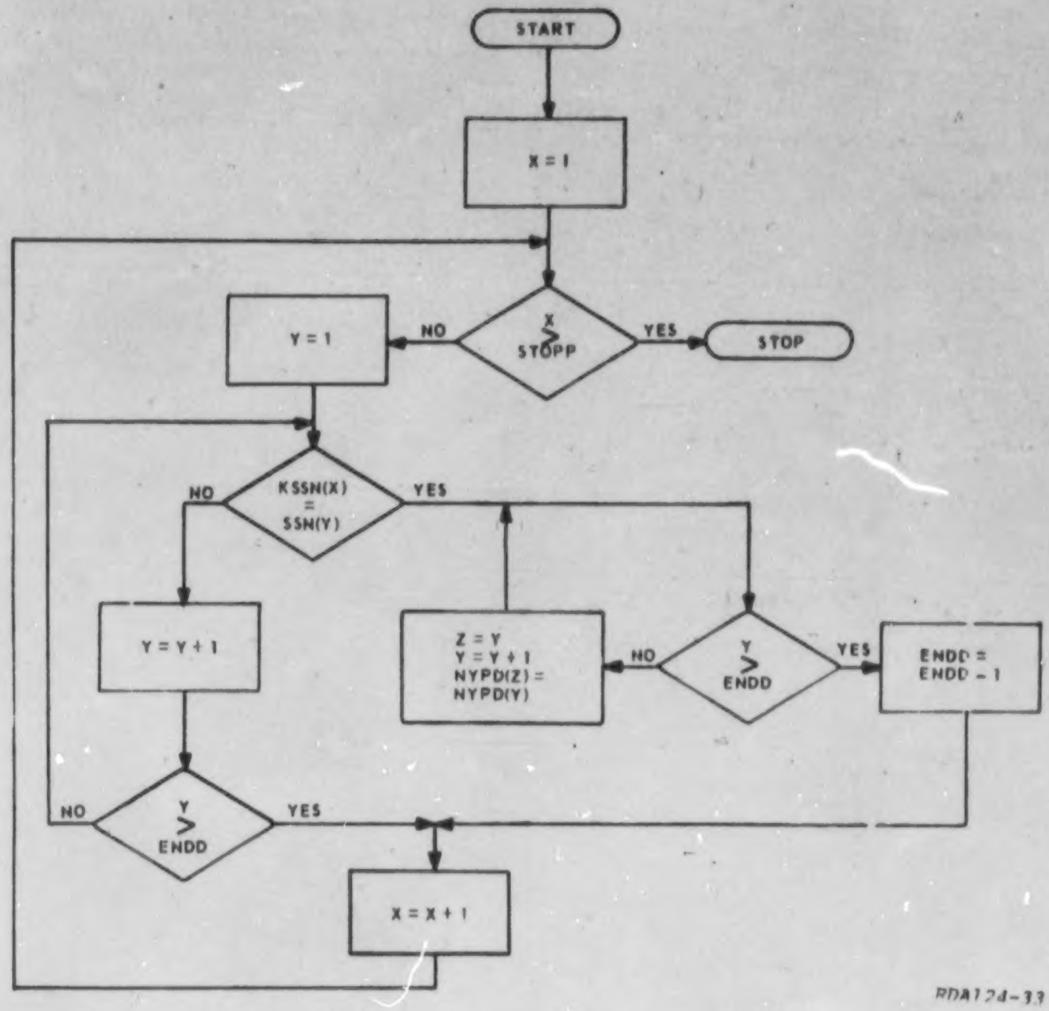
KNAME 1-20	KSSN 21-29

Solution 8

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
X = 1	1
Is X GT STOPP? Yes, go to 16. No, go to 3.	2
Y = 1	3
Is KSSN(X) = SSN(Y)? Yes, go to 5. No, go to 11.	4
Y = Y + 1	11
Is Y GT ENDD? Yes, go to 13. No, go to 4.	12
X = X + 1	13
Go to 2.	14
Is Y = ENDD? Yes, go to 6. No, go to 7.	5
ENDD = ENDD + 1	6
Go to 13.	15
Z = Y	7
Y = Y + 1	8
NYPD(Z) = NYPD(Y)	9
Go to 5.	10
Stop processing.	16

STEP THREE (Flowchart the sequence of operations.)

See figure 4-10.



RDA124-33

Figure 4-10. Deletion Flowchart (Multiple Deletion)

Merge Flowchart

A merge flowchart is essentially a modified insertion flowchart, which combines two ordered files into a third ordered file. Simply search through both files sequentially, building the new file in the desired order. The manner of search depends on the organization of both original files and the desired organization of the new file. The organization of the three files will determine the original setting of your counters, and the test values used. For example, if all three files are in ascending order, set the counters equal to 1 and test for the last record of a file (for all three files). If you have one file ascending, one descending, and want to produce an ascending new file, set the descending file counter equal to the total number of records in that file (testing for equal to zero), and set the other two ascending file counters equal to one (testing for the last record in the file).

Problem 9

MAIN is a file in memory with up to 1,800 records, each containing the following variables: NAME - customer name and BAL - the customer account balance. TOT1 is another variable which indicates the number of records in MAIN. BRANCH is a file in memory with up to 400 records, each containing the following variables: BNAME - customer name and BBAL - customer account balance. TOT2 is another variable which indicates the number of records in BRANCH. NUBANK is a new file in memory with up to 2,200 records, each containing the following variables: NUNAME and NUBAL. TTOT is another variable which indicates the number of records in NUBANK.

NAME 1-20	BAL 21-28

BNAME 1-20	BBAL 21-28

NUNAME 1-20	NUBAL 21-28

This problem concerns a main bank and branch bank which maintain separate small computers and bookkeeping sections. They decide to install one large computer at the main bank and an inquiry station at the branch. The separate files of the two banks must be merged into a new file in descending order based on key fields BAL and BBAL. At present, MAIN is in descending order and BRANCH is in ascending order.

Solution 9

STEP ONE (List the operations to be performed)	STEP TWO (Number the sequence of operations)
TTOT = TOT1 + TOT2	1
A = 1	2
C = 1	3
B = TOT2	4
Is C GT TTOT? Yes, go to 17. No, go to 6.	5
Is A GT TOT1? Yes, go to 8. No, go to 7.	6
Is B = 0? Yes, go to 13. No, go to 12.	7
Is BAL(A) GT BBAL(B)? Yes, go to 13. No, go to 8.	12

NUBANK(C) = MAIN(A) 13

A = A + 1 14

C = C + 1 15

Go to 5. 16

NUBANK(C) = BRANCH(B) 8

B = B - 1 9

C = C + 1 10

Go to 5. 11

Stop processing. 17

STEP THREE (Flowchart the sequence of operations.) (See figure 4-11.)

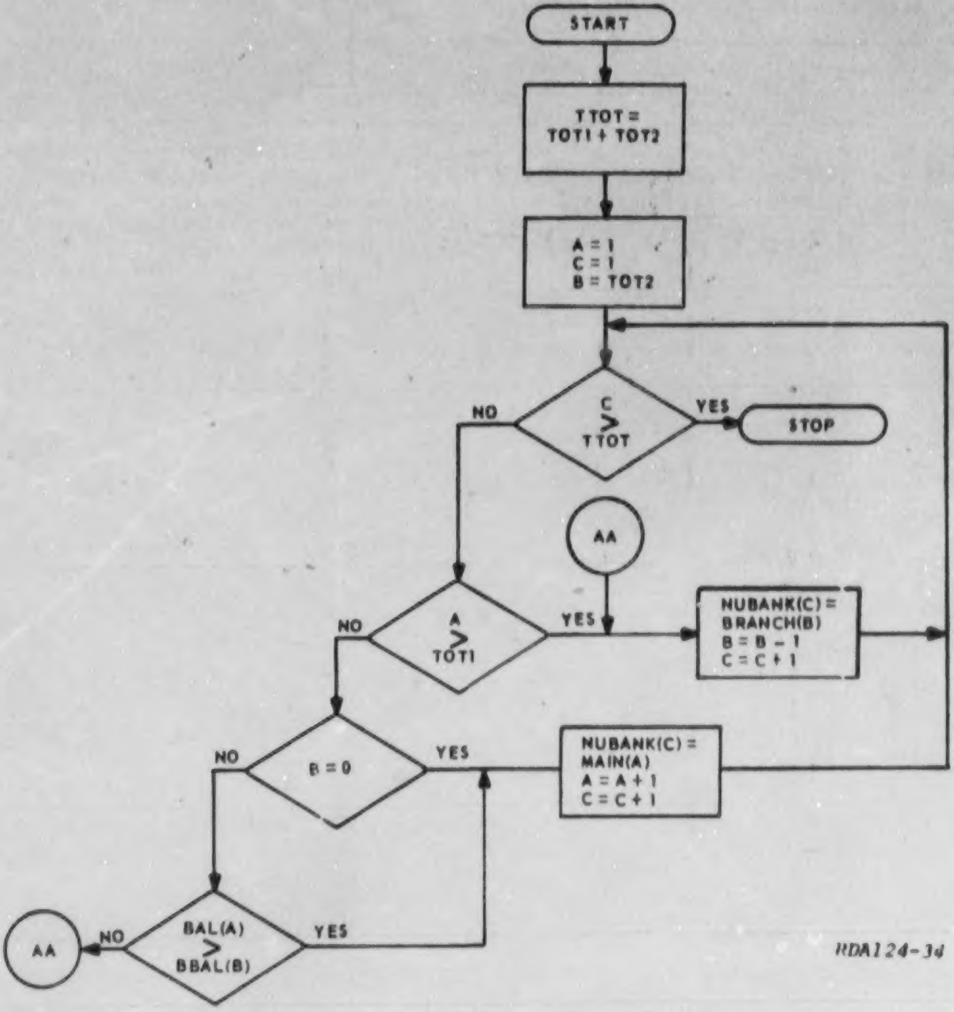


Figure 4-11. Merge Flowchart

FLOWCHART ANALYSIS

The analysis of a flowchart should yield three things:

1. A description of the operation shown in each block.

2. The purpose or what is accomplished by the flowchart.

3. The form of the data at any specified point and what form of data will result in branching to each output from a decision block. Since you have already drawn several flowcharts, you may be able to analyze the following flowcharts with little difficulty. In order to obtain the most help from this section, you should NOT read the explanations until after you have studied the flowchart and made a decision on how you would explain it. Then, you should read the explanation and see if your explanation agrees. An explanation will also be given for the general purpose of that flowchart. Note that each block of the flow is identified with a letter; these letters are used to reference the explanations.

STU is a file in memory with up to 100 records, each containing the following variables: NAME and PCG. ENDD is another variable which indicates the number of records in STU. (See Figure 4-12 below.)

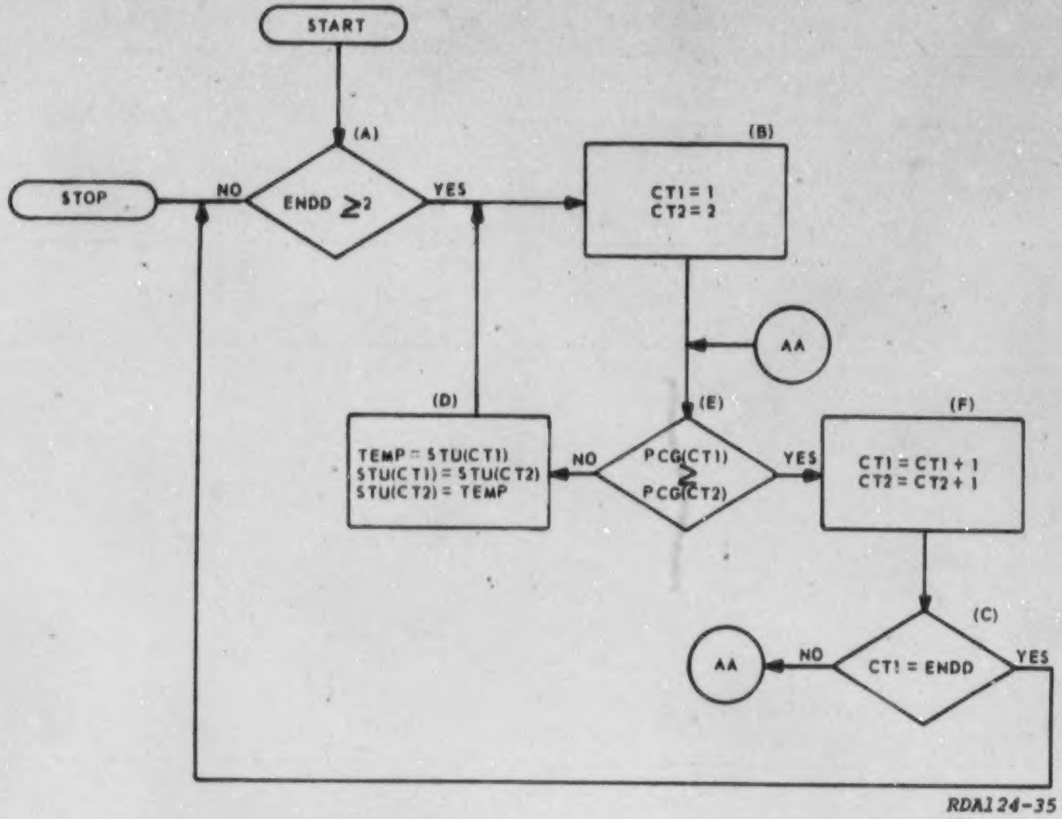


Figure 4-12. Flowchart 1

Analysis of Flowchart 1

Block (A) prevents attempting to sort a file with fewer than two records. If ENDD is equal to 0 or 1 the flow will stop. If ENDD is equal to 2 through 100, it will branch to Block (B).

Block (B) initializes the counters on each exchange loop. CT1 and CT2 are initialized to 1 and 2, respectively.

Block (C) tests the value of the counter each time two consecutive records are found to be in the desired order. It branches to STOP after the next to the last record has been compared to the last record and found to be in the desired order. If ENDD is equal to 100 and CT1 is equal to 1 through 99, it will branch to Block (E) and compare the next two records. If ENDD and CT1 are both equal to 100, it will stop.

Block (D) exchanges the values of two consecutive records to obtain the desired order.

Block (E) compares the value of PCG in each record against the value of PCG in the next sequential record. If the lower numbered record has a lower value, it will branch to Block (D). If the lower numbered record has an equal or higher value, it will branch to Block (F).

Block (F) increments the counters when two records are found to be in the desired order.

The overall purpose of Flowchart 1 is to sort file STU into descending order, based on values in key field PCG.

Analysis of Flowchart 2

Block (A) prevents attempting to insert a new record into a full file. If STOPP is equal to 300, it will branch to Block (B). If STOPP is equal to 1 through 299, it will branch to Block (C).

Block (B) writes "File Full" on the line printer to inform the operator that CAR already contains its maximum number of records.

Block (C) initializes the counter.

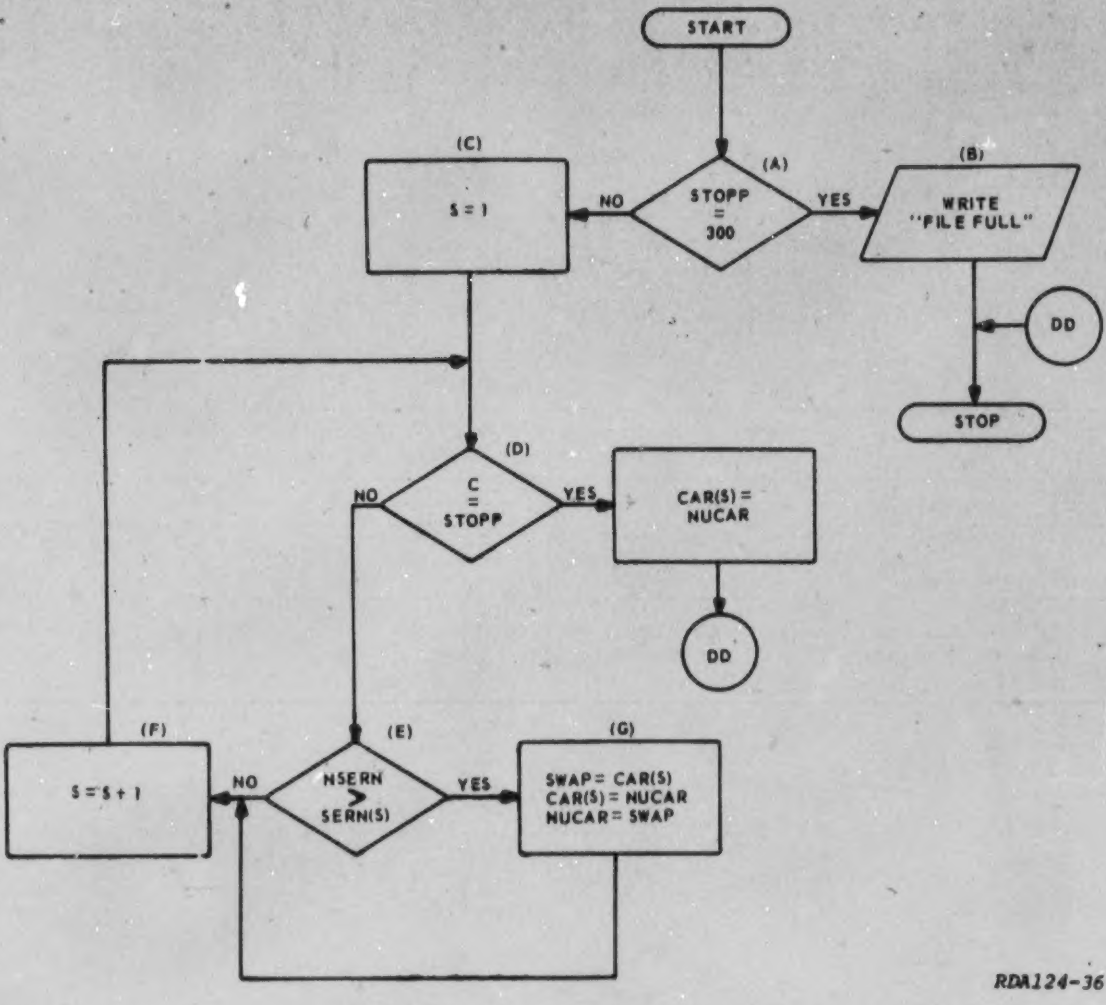
Block (D) compares the counter (which counts the loops) with STOPP to insure that we do not continue exchanging records after we reach the end of CAR.

Block (E) compares the values of NSERN and SERN. It branches to Block (H) if NSERN is smaller than SERN and branches to Block (F) if NSERN is equal to or greater than SERN.

Block (F) increments the counter.

Block (G) inserts the record into the first slot in the file that contains no meaningful data and updates STOPP to reflect the new last record in CAR. Since only one record is to be inserted, the flowchart stops.

The overall purpose of this flowchart is to insert the record NUCAR into file CAR at its proper location.



RDA124-36

Figure 4-13. Flowchart 2

FLOWCHART CORRECTION

Due to the transfer of personnel, one programmer may start on the solution to a problem and another be assigned to complete the program. Also, it is common practice to ask for help in debugging a program because familiarity with the solution sometimes causes the originator to overlook errors another person can readily see. For whatever reason, there will be times when you are required to analyze a flowchart that someone else constructed and find one or more errors.

Many times errors are easily spotted, but if they cannot be located, the following procedure should be of value.

1. Analyze the problem as if you were going to construct a flowchart and a program.

2. Analyze the flowchart that was designed to solve the problem.

3. Compare the problem analysis with the flowchart analysis and check off each operation (make your checkmarks on the problem analysis). During this comparison, look for index registers being incremented instead of decremented or vice versa, incorrect value being used for testing a loop, index register not being initialized at the proper time, yes and no legs reversed, exchanging on equal (EQ, GQ, LQ), etc.

4. Determine if the problem analysis contains one or more operations not shown in the flowchart analysis, thus indicating omission from the flowchart and the program. (This will be revealed by the absence of a checkmark beside the operation.)

Flowchart analysis and flowchart corrections are done concurrently in most instances. In fact, you cannot expect to correct logic unless you fully understand the problem being solved. In conclusion to this discussion, one point must be restated--there are always two or more correct solutions to a problem. That is why, when you correct someone else's work, comparing his flowchart with your problem analysis offers the best chance of combining your efforts to produce a successful program.

APPENDIX A
(Extract from Chapter 6 of AFM 171-10, Vol I)

Chapter 6
FLOWCHART SYMBOLS FOR DATA PROCESSING

020601. **General.** The purpose of this chapter is to establish flowchart symbols for use in the preparation of flowcharts for automatic data processing systems and applications. These symbols are the American Standard Flowchart Symbols which were approved by the Department of Defense.

020602. **Responsibility.** It is mandatory that the American Standards Association symbols be used by the Air Force in the preparation of all new and revised ADPS flowcharts. Existing flowcharts need not be reaccomplished for the sole purpose of converting to the American Standards Association symbols.

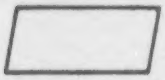
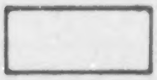
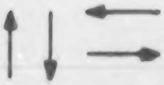
020603. **Flowchart Symbols.**

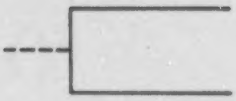
a. Symbols Represent Functions. Symbols are used on a flowchart to represent the functions of a data processing system. These functions are INPUT/OUTPUT, PROCESSING, FLOW DIRECTION, and ANNOTATION.

A basic symbol is established for each function and can always be used to represent that function. Specialized symbols are established which may be used in place of a basic symbol to give additional information.

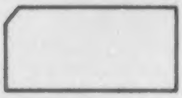
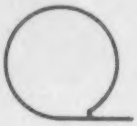
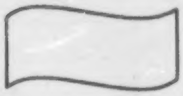
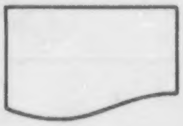
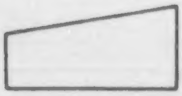
The size and the dimensional ratio of each symbol may vary depending on its specific use but not to the point of losing its identity.

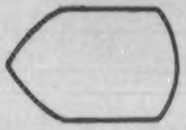
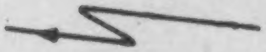
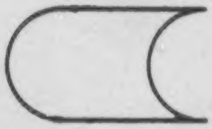
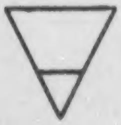
b. Basic Symbols.

Symbols	Descriptions
	Input/Output Symbol. The symbol shown represents the input/output function (I/O); i.e., the making available of information for processing (input) or the recording of processed information (output).
	Processing Symbol. The symbol shown represents the processing function; i.e., the process of executing a defined operation or group of operations resulting in a change in value, form, or location of information, or in the determination of which of several flow directions are to be followed.
	Flow Direction Symbol. The symbols shown represent the flow direction function; i.e., the indication of the sequence of available information and executable operations. Flow direction is represented by lines drawn between symbols. Normal direction flow is from top to bottom and left to right. When the flow direction is not top to bottom and left to right, open arrowheads shall be placed on reverse

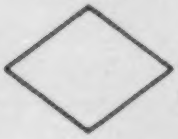
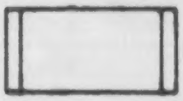
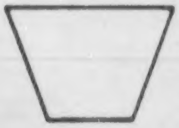
Symbols	Descriptions
	direction flowlines. When increased clarity is desired, open arrowheads can be placed on normal direction flowlines. When flowlines are broken due to page limitation, connector symbols shall be used to indicate the break. When flow is bidirectional, it can be shown by either single or double lines but open arrowheads shall be used to indicate both normal direction flow and reverse direction flow.
	Annotation Symbol. The symbol shown represents the annotation function; i.e., the addition of descriptive comments or explanatory notes as clarification. The broken line may be drawn either on the left as shown or on the right. It is connected to the flowline at a point where the annotation is meaningful by extending the broken line in whatever fashion is appropriate.

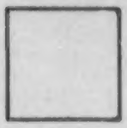
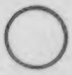
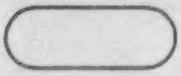
c. Specialized Input/Output Symbols. Specialized I/O symbols may represent the I/O function and, in addition, denote the medium on which the information is recorded or the manner of handling the information or both. If no specialized symbol exists, the basic I/O symbol is used. These specialized symbols are:

Symbols	Descriptions
	Punched Card Symbol. The symbol shown represents an I/O function in which the medium is punched cards, including mark sense cards, partial cards, stub cards, etc.
	Magnetic Tape Symbol. The symbol shown represents an I/O function in which the medium is magnetic tape.
	Punched Tape Symbol. The symbol shown represents an I/O function in which the medium is punched tape.
	Document Symbol. The symbol shown represents an I/O function in which the medium is a document.
	Manual Input Symbol. The symbol shown represents an I/O function in which the information is entered manually at the time for processing, by means of online keyboards, switch settings, pushbuttons, card readers, etc.

Symbols	Descriptions
	Display Symbol. The symbol shown represents an I/O function in which the information is displayed for human use at the time of processing, by means of online indicators, video devices, console printers, plotters, etc.
	Communication Link Symbol. The symbol shown represents an I/O function in which information is transmitted automatically from one location to another. To denote the direction of data flow, the symbol is always drawn with superimposed arrowheads.
	Online Storage Symbol. The symbol shown represents an I/O function utilizing auxiliary mass storage of information that can be accessed online; e. g. , magnetic drums, magnetic disks, magnetic tape strips, automatic magnetic card systems or automatic microfilm chip or strip systems.
	Offline Storage Symbol. The symbol shown represents any offline storage of information, regardless of the medium on which the information is recorded.

d. Specialized Processing Symbols. Specialized processing symbols may represent the processing function and, in addition, identify the specific type of operation to be performed on the information. If no specialized symbol exists, the basic processing symbol is used. These specialized symbols are:

Symbols	Descriptions
	Decision Symbol. The symbol shown represents a decision or switching type operation that determines which of a number of alternate paths is to be followed.
	Predefined Process Symbol. The symbol shown represents a named process consisting of one or more operations or program steps that are specified elsewhere; e. g. , subroutine or logical unit.
	Manual Operation Symbol. The symbol shown represents any offline process geared to the speed of a human being.

Symbols	**Descriptions**
	**Auxiliary Operation Symbol.** The symbol shown represents an offline operation performed on equipment not under direct control of the central processing unit.
	**Connector Symbol.** The symbol shown represents a junction in a line of flow. A set of two connectors is used to represent a continued flow direction when the flow is broken by any limitation of the flowchart. A set of two or more connectors is used to represent the function of several flowlines with one flowline or the junction of one flowline with one of several alternate flowlines.
	**Terminal Symbol.** The symbol shown represents a terminal point in a system or communication network at which data can enter or leave; e.g., start, stop, halt, delay, or interrupt.

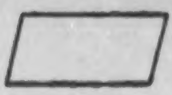
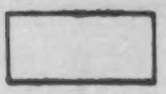
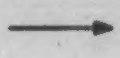
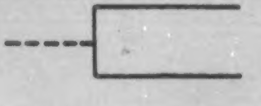
f. Existing flowchart templates, i.e., those provided by the manufacturers, may be utilized to form the flowchart symbols above.

020604. Summary of American Standard Flowchart Symbols.

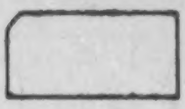
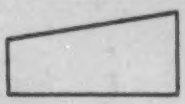
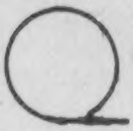
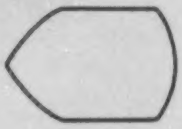
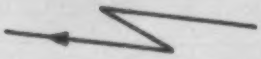
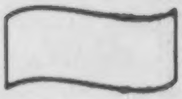
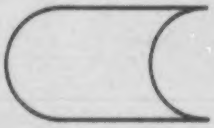
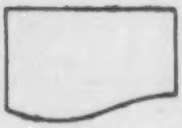
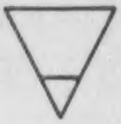
A summary of flowchart symbols is illustrated on the following page.

SUMMARY OF FLOWCHART SYMBOLS

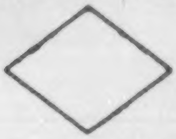
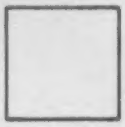
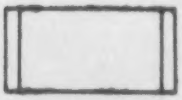
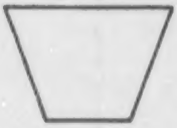
BASIC SYMBOLS

INPUT/OUTPUT	PROCESSING	FLOW DIRECTION	ANNOTATION
			

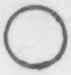
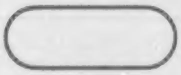
SPECIALIZED INPUT/OUTPUT SYMBOLS

PUNCHED CARD		MANUAL INPUT	
MAGNETIC TAPE		DISPLAY	
		COMMUNICATION LINK	
PUNCHED TAPE		ONLINE STORAGE	
DOCUMENT		OFFLINE STORAGE	

SPECIALIZED PROCESSING SYMBOLS

DECISION		AUXILIARY OPERATION	
PREDEFINED PROCESS		MANUAL OPERATION	

ADDITIONAL SYMBOLS

CONNECTOR		TERMINAL	

APPENDIX B

STORED PROGRAM INSTRUCTIONS

Every problem a computer handles, from adding a column of figures to analyzing cosmic rays, must first be broken down by a human programmer into simple steps that the computer can solve with its yes-no language of binary notation.

With early computers, the programmer had to set up these steps by plugging wires into holes in the computer's problem board, in a manner similar to the wiring of the IBM 557 Interpreter. These wires established a "route of reasoning" along which the problem traveled through the interpreter. By manipulating the wires, the programmers were able to choose different routes for different problems. However, even simple problems frequently required hours of painstaking setup time for each program to be run.

With the advent of practical, main-storage devices, however, the picture changed radically. It became possible to write instructions for the computer and to store these instructions in the main storage unit. This set of instructions was called a program.

Since these instructions could be keypunched on cards and then loaded quickly into core, converting the computer workload from one problem to another became a quick and easy procedure. This gave the computer an almost unlimited flexibility. It allowed the computer to be applied to a great number of different procedures by simply reading in, or loading, the proper program into memory.

The Stored Program

A program is the complete plan for the solution of a problem by a computer, including the complete sequence of machine instructions necessary to solve the problem. A stored program is this series of instructions stored internally in the computer, directing the step-by-step operations of the machine. These groups of instructions taken as a whole are a program and will produce the desired output from the machine.

The program is loaded into memory, and once loaded the computer turns control over to the program. By using the various instructions, the program calls in data from the input devices, processes it, and sends the results to output devices. The program resides in memory along with the data. Figure B1 shows the memory layout.

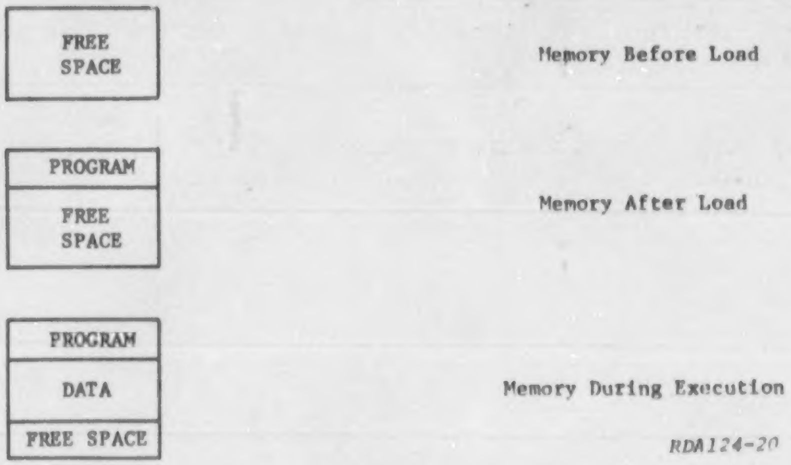


Figure B1

Types of Instructions

Each computer has its own set of instructions which the machine is capable of performing. The instruction set differs from manufacturer to manufacturer and even between machines by the same manufacturer. Despite these differences, all computer instructions fall into one of five basic divisions: arithmetic, move, compare, branch, and input-output.

Arithmetic Instructions

An arithmetic instruction causes the computer to perform the specified mathematical operation such as add, subtract, multiply, or divide on the data at the locations given in the instructions.

Move Instructions

The move instruction causes the computer to transfer data from one location to another in memory. Upon execution of a move instruction, the contents of the location in memory from which the data is moved remains unchanged. During the move, the contents of the receiving memory location is destroyed and replaced by the contents of the sending location. This is known as "destructive read-in" and "nondestructive read-out."

Compare Instructions

The compare instructions cause the computer to compare the data stored in one core location with the data stored in a second core location. This comparison causes an internal compare indicator to be set to the results of the compare. If the first location were high and the second low, the compare indicator would be set to high. If the first location and second location were equal, the compare indicator would be set to equal. But, if the first location were low and the second location high, the compare indicator would be set to low. Thus, the compare indicator can show three results of a compare instruction: high, equal, or low.

Branch Instruction

In normal operations the computer performs groups of instructions sequentially, one after another. Occasionally it may become necessary to alter this normal sequence and execute some other sequence of special instructions. Or it may be necessary to repeat a group of instructions several times. To do this it is necessary for the computer to execute a branch instruction telling it to go to another location in memory for its next instruction.

There are two types of branches: unconditional and conditional branches. The unconditional branch simply tells the computer to go automatically to another group of instructions and begin executing them. The conditional branches allow the computer to make decisions whether to execute another group of instructions or to continue executing, in sequence, the instructions immediately following. The machine makes decisions by interrogating the compare indicator for specific conditions. There are several types of conditional branches which include branch or compare indicator high (refer back to compare instructions), branch or compare indicator equal, and branch or compare indicator low. These branches are very important for they allow the computer to execute groups of instructions repetitively and to make logical decisions as to courses of action for the computer to take.

Input-Output Instruction

In order for the machine to process data it must be able, on instruction or command, to obtain this data from external sources. This is done through input instructions to various peripheral devices such as card readers. These instructions bring data into primary storage for processing. To relay its data on to us the machine must make use of output instructions. Typical output instructions may print the results of calculations on a printer, punch them on a Hollerith card, or write them to magnetic tape.

Instruction Formats

Instruction formats of any computer may be divided into two distinct parts. The first part of the instruction will consist of the command to be performed. This is the operation of the instruction. This operation is usually coded in some way to make it legible to the computer and to the programmer. Since it is an operation, and it is coded, it is usually referred to as the operation code or, in abbreviated form, the op code.

Almost all operation codes require that some action be performed on data in internal storage. The op code therefore requires an object on which to operate. This object will generally be data which will have an address giving its exact location in internal storage. Thus, the second part of the instruction is known as the operand or address and usually contains the address of the data that is to be accessed. The basic instruction format of a computer is the op code and operand (address).

Single Address Format

There are many variations of the basic instruction format. Starting with the simplest and working up to the more complex, the first format is the single address format.

OP CODE	ADDRESS

RDA124-21

Figure B2. Single Address Format

With the single address format, an operation will be performed in one location in storage. This presents a wide variety of problems. For example, to transfer data to a location in storage, the data must be moved from one location in storage to another. How can this be accomplished when only using one address? If one location in storage is to be added to another, how can the two locations be indicated with only one address?

To solve these problems, registers and accumulators must be used. The register is a special-purpose storage location or a part of the machine circuitry itself. The word "register" applies to a broad group of devices which have many purposes. Specific registers are given names depending upon their use. An instruction register holds the program instruction that the processor is currently executing. An address register holds the address of the operand specified by the instruction. An accumulator is a register to form sums and other arithmetic results for single-address computers that handle one operand at a time.

These accumulators and registers provide the intermediate step necessary to perform move and arithmetic operations with the single-address format. A disadvantage of this method is that two instructions are required to perform one operation.

Two-Address Format

To cope with the objection of using two instructions when only one is really necessary, some machines use a two-address instruction format. With this format, the need for intermediate registers is eliminated, and the power of each instruction is almost doubled. The general format for this type of instruction is shown in figure B3.

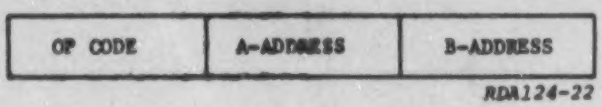


Figure B3. Two-Address Format

For ease of programming, it seems much more logical to use the two-address machine since the computer is now performing one action on two things, i.e., move this to that, add this to that, etc. This eliminates the need for intermediate steps using registers and thereby cuts down on the number of steps that must be written. However, here the programmer is dealing with two areas of storage and this may be difficult at debugging time when attempting to trace through the written program. Also, the data in one address is destroyed by the storage of the result of the operation.

Three-Address Format

The last format presented is actually a takeoff to all other variations of instruction formats. The final format is that of a three-address machine. The real versatility of this type machine is demonstrated in the arithmetic operations. In a normal add instruction, the A-field is added to the B-field and the result is placed in the C-field. This takes care of the objection raised in the previous sections where the contents of one of the operands was destroyed by the result of the operation.

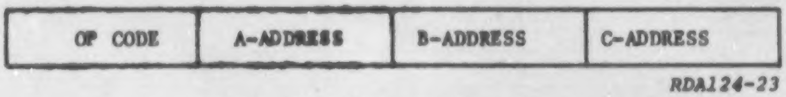


Figure B4. Three-Address Format

Obviously the formats presented are not the only ones used by all manufacturers, but the formats used will be variations of the three basic ones shown.

The limitations and advantages of each instruction format are the important points here. When looking at any machine, one immediate question should be: What is the format of the instruction? This will indicate the answer to several questions which the programmer must know. How difficult is the machine to program? What can be accomplished with each instruction? How many steps would be necessary to program a problem? While the format itself does not answer these questions definitely, it does indicate generally what the answers may be.

The power and functions of an instruction can rarely be evaluated merely by determining the normal instruction format. This can only be done after a programmer has programmed machines using various formats.

Fixed and Variable Word Length

We have discussed different methods used in addressing data and instructions and we know that an instruction has two basic parts called the op code and operand. It is obvious that both instructions and data consume a certain quantity of storage. What are some of the factors that determine the size of instructions and data in internal storage? To answer this question, it is necessary to define the terms fixed word and variable word length computers.

Fixed Word Length Computers

If a machine is a fixed word length computer, it will automatically manipulate the same number of locations for every operation. Thus when adding two fields in a 6-digit fixed length computer, the machine will add two 6-position fields of data together. If one data field is moved to another, six digits will be moved to replace six other digits. If one field of data is compared to another, one 6-position field of data will be compared to another 6-position field.

Most computers which have fixed length data fields can usually manipulate multiples of the fixed length. Thus in the examples given, the machine could be moving, adding and comparing 12, 18, 24, etc., positions of storage. This would also require a method of telling the machine the number of words being operated on at one time.

If the data fields do not correspond to the fixed format of the machine, it is wasting valuable storage positions. Thus, if the data field is 14 positions and the fixed word length is 6, four positions of the third word used would be wasted.

If a machine has fixed data length, it may also have fixed instruction length. These computers usually have a few instructions that do not require the use of an operand. It is obvious that these positions are going to waste when used in an operation which does not utilize all of the instruction format. This is one of the disadvantages of a fixed length machine. However, most machines have methods that enable the unused portion of an instruction or data storage location to be "packed" with other data in order to reduce the amount of wasted storage.

What then are the advantages of using a fixed word machine? First, it is easier to keep track of the address of certain instructions in storage and also of data fields since everything will be in multiples of the fixed length. In an 8-position fixed length instruction word, each instruction will take 8 positions of storage. Thus, if the first instruction is in positions 6-13, the fifth instruction would be in locations 38-45 and its address would be location 38. Remember that instructions are addressed by the high order position.

The second, and most important, advantage of the fixed word computer is its ability to transmit data in parallel lines. This is the ability of the computer to send or manipulate data in fixed groups of digits. For example, if five positions of storage were moved from one location to another on a fixed word machine, all five would move at one time rather than one digit at a time. This occurs because the manufacturer provides enough transmission paths to transmit one complete computer word of storage at a time.

Variable Word Length Computers

Not all machines are fixed length computers. The other major breakdown to be considered is variable length machines. In this type of machine, there is no limit to the amount of data which may be contained on one data word or in one instruction in storage. This does not mean that the words or instructions are unlimited in size, but does mean that the word can contain as little as one digit and increase in size until the limits of the circuitry are reached.

APPENDIX C

Segmenting Structured Programs

Imagine a 100-page program written in structured code. Although it is highly structured, such a program is still not very readable. The extent of a major DO loop may be 50 or 60 pages, or an IFELSE statement may take 10 or 15 pages. This is simply more than the eye can comfortably take in or the mind retain for the purpose of programming.

However, with our program in structured form, we can begin a process, which we can repeat over and over until we get the whole program defined. This process is to formulate a 1-page skeleton program which represents that 100-page program.

We do this by selecting some of the most important lines of code in the original program and then filling in what lies between those lines by names. Each new name will refer to a new segment to be stored in a library and called by a macro facility insert. In this way, we produce a program segment with something under 50 lines, so that it will fit on one page. This program segment will be a mixture of control statements and macro calls with possibly a few initializing, file, or assignment statements as well.

The programmer must use a sense of proportion and importance in identifying what is the forest and what are the trees out of this 100-page program. It corresponds to writing the "high level flowchart" for the whole program, except that a completely rigorous program segment is written. A key aspect of Structured Programming is that any segment referred to by name, control enters at the top and exits at the bottom, and has no other means of entry or exit from other parts of the program. Thus, when reading a segment name, at any point, the reader can be assured that control will pass through that segment and not otherwise affect the control logic on the page he is reading.

In order to satisfy the segment entry/exit requirement, we need only be sure they include all matching control logic statements on a page. For example, the ENDO to any DO, and the ELSE to any IF should be put in the same segment.

For the sake of illustration, this first segment may consist of some 30 control logic statements, such as DOWHILE's, IFELSE's, perhaps another 10 key initializing statements, and some 10 macro calls. These 10 macro calls may involve something like 10 pages of programming each for the original 100 pages, although there may be considerable variety among their sizes.

Now we can repeat this process for each of these 10 segments. Our end result is a program which has been organized into a set of named member segments, each of which can be read from top to bottom without any side effects in control logic, other than what is on that particular page. A programmer can access any level of information about the program, from highly summarized at the upper level segments to complete details in the lower levels.

In the preceding paragraphs, we assumed that a large structured program somehow existed, already written with structured control logic, and discussed how we could conceptually reorganize the identical program in a set of more readable segments. In this following text, we observe how we can create such structured programs a segment at a time in a natural way.

Creating a Structured Program

We suppose that a program has been well designed and that we are ready to begin coding. We also note a common pitfall in programming is to "lose our cool"--i.e., begin coding before the design problems have been thought through well enough. In this case, it is easy to compromise a design because code already exists which isn't quite right, but "seems to be running correctly"; the result is that the program gets warped around code produced on the spur of the moment.

Our main point is to observe that the process of coding can take place in practically the same order as the process of extracting code from our imaginary large program in the previous section. That is, armed with a program design, one can write the first segment which serves as a skeleton for the whole program, using segment names, where appropriate, to refer to code that will be written later. In fact, by simply taking the precaution of inserting dummy members into a library with those segment names, one can compile or assemble, and even possibly execute this skeleton program, while the remaining coding is continued. Very often, it makes sense to put a temporary statement "got to here OK" as a single executable statement in such a dummy member.

Now, the segments at the next level can be written in the same way, referring as appropriate to segments to be later written and setting up dummy segments as they are named in the library. As each dummy segment becomes filled in with its code in the library, the recompilation of the segment that includes it will automatically produce updated, expanded versions of the developing program. Problems of syntax and control logic will usually be isolated within the new segments so that debugging and checkout goes correspondingly well with such problems so isolated.

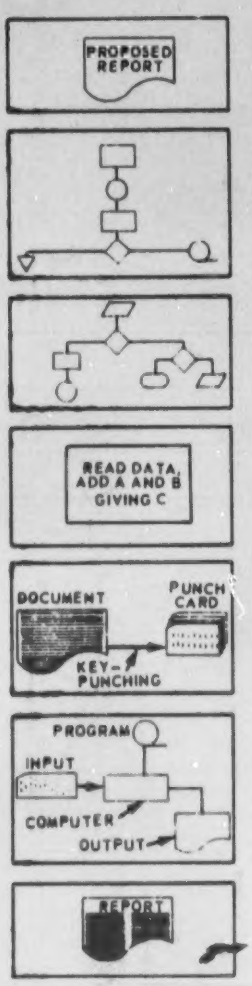
It is clear that the programmer's creativity and sense of proportion play a large factor in the efficiency of this programming process. The code that goes into earlier sections should be dictated, to some extent, not only by general matters of importance, but also questions of getting executable segments reasonably early in the coding process. For example, if the control logic of a skeleton module depends on certain control variables, their declarations and manipulations may want to be created at fairly high levels in the hierarchy. In this way, the control logic of the skeleton can be executed and debugged, even in the still skeleton program.

Note that several programmers may be engaged in the foregoing activity concurrently. Once the initial skeleton program is written, each programmer could take on a separate segment and work somewhat independently within the structure of an overall program design. The hierarchical structure of the programs contribute to a clean interface between programmers. At any point in the programming, the segments already in existence give a concise framework for fitting in the rest of the work.

APPENDIX D

STEPS IN PROGRAM PROBLEM SOLVING

To successfully handle a data processing problem, the problem must be attacked in an orderly, step-by-step fashion. This cycle includes all the procedures necessary to complete a computer program. These steps are adhered to in developing every successful computer system because this logical sequence of operations assures that each program is created in the shortest amount of time, with the fewest possible errors. A description of the steps follows (see figure D1).



STEPS	COMMENTS
1 ANALYSIS OF A PROBLEM	COLLECT INFORMATION AND DECIDE WHAT INFORMATION IS NEEDED, FREQUENCY OF PROCESSING, ETC.
2 DESIGN OF A SYSTEM TO PROVIDE THE INFORMATION	PLAN THE SYSTEM OF PROCESSING USING A SYSTEM FLOWCHART PLUS LAYOUTS OF REPORTS, DOCUMENTS, RECORDS, FILES, ETC.
3 PLANNING THE COMPUTER LOGIC	PLAN THE PROGRAM LOGIC USING A PROGRAM FLOWCHART OR DECISION TABLE.
4 PROGRAM PREPARATION	WRITE THE PROGRAM OF INSTRUCTIONS AND DEBUG IT TO REMOVE ALL TEXTUAL OR LOGICAL ERRORS. TRANSLATE IT INTO MACHINE LANGUAGE. PREPARE DOCUMENTATION.
5 INPUT DATA PREPARATION	PREPARE INPUT DATA BY COLLECTING OR TRANSCRIBING DATA INTO MACHINE-READABLE FORM SUCH AS PUNCHED CARDS.
6 RUNNING OF PROGRAM	PUT PROGRAM INTO COMPUTER MEMORY. COMPUTER PROGRAM READS DATA, PROCESSES IT, AND OUTPUTS THE RESULT.
7 USE OF THE OUTPUT	

RDA124-9

Figure D1. Steps in Problem Solving

1. Analysis of the Problem

Obviously, the first step must be to determine exactly what must be accomplished. Completing the problem definition is often the most time-consuming step. Also, it must be decided what type of data will be used, what operations must be performed on the data, and what type of result is required.

2. Design of System Flowchart

The processing system must be laid out. Usually a system flowchart is used to describe the interrelationship of the individual programs, and to show the origins and destinations of data. Report layouts are usually determined at this point.

3. Detail Flowchart

Once the problem has been thoroughly defined and then broken down into subsections (System Flowchart), the process the computer will use in solving the problems should be determined. The most effective method for doing this is to diagram the logical, step-by-step solution using the operations the computer is capable of performing.

4. Program Preparation

Naturally, the program must be in a form acceptable by the computer. Translating the data formats and the flowchart procedures into an acceptable computer language is called coding. Although every program must be coded, the language used may vary from computer to computer, and application to application.

The coded program should be checked over carefully before it is run on the computer (desk-checking) to eliminate errors. These errors may be the programmer's own logic or coding errors or keypunch errors by the keypunch operator. Since computer time is limited and extremely expensive, it is advantageous to keep to a minimum the amount of time spent testing and rerunning the program. By reducing the amount of coding errors, the amount of time needed for testing and rerunning can also be reduced.

After the program has had a thorough desk-check, it is ready to be translated from the programming language to the machine language. This translation is made by the computer, using special programs which will be covered later in this chapter.

After the program has been written, desk-checked, and translated into machine language, it is ready for testing. Testing is necessary because minor errors in flowcharting and coding may not have been detected. The first test should be made with prepared test data that will force the execution of every program instruction. This will not normally happen if actual live data is used. The test data should contain every possible condition, whether or not the condition represents a legitimate occurrence. The test data should be included with the final program documentation. The testing, or program checkout, of a new or revised program is referred to as debugging. The purpose of the test is to detect and correct any errors made during programming.

Documentation is a continuous part of programming. While developing a program, precise records should be kept on all information pertaining to the program. These records include record layouts, flowcharts, coding, and all modifications.

Included in the documentation should be a general description of the program, written in layman terms. This will enable users of a program and management to understand the essentials about the program. It should present the function of each program, any option the program features, types of input-output, the language the program was written in, and the machine for which it is written. If the program is a scientific or an engineering oriented program, some information should be given about the method used in solving the problem.

Instructions should always be included for the operations section. If the program needs some special console switch settings, these must be provided to the operator. Instructions regarding special input-output setup procedures are needed. Such items as special carriage control tapes, printer forms, special cards, and tape names are just some of the information operators need about each job. Documentation about exceptions as well as normal operations should be given, so that anything the program might do will not come as a surprise to the operators.

Flowcharting is the main form of program documentation; however, there are other pieces of documentation used with programs. Good documentation on input-output formats is required, especially for record and file layouts of disk and tape. Included in any documentation package should be any modifications or changes to the program. Also, a complete listing of the program instructions is mandatory for use when further modifications to the program are required.

In the Air Force environment, documentation is extremely important because of the inevitable turnover of personnel. Good documentation will insure that no one person is indispensable.

5. Input Data Preparation

The conversion of source documents into a suitable input medium is normally the responsibility of the individual user agency. Using the formats provided in the system documentation, they will keypunch the information onto cards and then turn the decks into the data processing installation (DPI) at the proper times.

It is the responsibility of the DPI to integrate the program into its production schedule and see to it that the program is processed on time.

6. Running of the Program

Once the program has been properly translated, and the data has been keypunched on cards (or converted to magnetic tape), the program is entered into core. It calls in the data; processes it; and, unless a malfunction occurs, produces the final report.

It is the responsibility of the operator to ensure that the program runs correctly. He will be required to handle minor problems as they occur and keep the computer running properly by using standard recovery procedures and program documentation.

7. Use of the Output

The output is, of course, the purpose of running the program in the first place. Any printer output should be inspected by the operator for completeness or obvious errors before being turned over to the user agencies. Newly created tapes should be properly labeled and stored in the tape library. These and additional procedures are the responsibility of the DPI's Production Control Section.

APPENDIX E

STANDARD CHARACTER SET

Standard Character Set	BCD (Binary)	BCD (Octal)	Hollerith Card Code	ASCII Code	UASCI Code	EBCDIC Code
0	000000	00	0	060	060	360
1	000001	01	1	061	061	361
2	000010	02	2	062	062	362
3	000011	03	3	063	063	363
4	000100	04	4	064	064	364
5	000101	05	5	065	065	365
6	000110	06	6	066	066	366
7	000111	07	7	067	067	367
8	001000	10	8	070	070	370
9	001001	11	9	071	071	371
[	001010	12	2-8	133	133	
#	001011	13	3-8	043	043	173
@	001100	14	4-8	100	100	174
:	001101	15	5-8	072	072	172
>	001110	16	6-8	076	076	156
?	001111	17	7-8	077	077	157
b̸	010000	20	(blank)	040	040	100
A	010001	21	12-1	141	101	301
B	010010	22	12-2	142	102	302
C	010011	23	12-3	143	103	303
D	010100	24	12-4	144	104	304
E	010101	25	12-5	145	105	305
F	010110	26	12-6	146	106	306
G	010111	27	12-7	147	107	307
H	011000	30	12-8	150	110	310
I	011001	31	12-9	151	111	311
&	011010	32	12	046	046	
.	011011	33	12-3-8	056	056	113
]	011100	34	12-4-8	135	135	
(	011101	35	12-5-8	050	050	115
<	011110	36	12-6-8	074	074	114
\	011111	37	12-7-8	134	134	
↑	100000	40	11-0	136	136	
J	100001	41	11-1	152	112	321
K	100010	42	11-2	153	113	322
L	100011	43	11-3	154	114	323
M	100100	44	11-4	155	115	324
N	100101	45	11-5	156	116	325
O	100110	46	11-6	157	117	326
P	100111	47	11-7	160	120	327
Q	101000	50	11-8	161	121	330
R	101001	51	11-9	162	122	331
-	101010	52	11	055	055	140
$	101011	53	11-3-8	044	044	133
*	101100	54	11-4-8	052	052	134
)	101101	55	11-5-8	051	051	135
;	101110	56	11-6-8	073	073	136
'	101111	57	11-7-8	047	047	175
+	110000	60	12-0	053	053	116
/	110001	61	0-1	057	057	141
S	110010	62	0-2	163	123	342
T	110011	63	0-3	164	124	343
U	110100	64	0-4	165	125	344
V	110101	65	0-5	166	126	345
W	110110	66	0-6	167	127	346
X	110111	67	0-7	170	130	347
Y	111000	70	0-8	171	131	350
Z	111001	71	0-9	172	132	351
←	111010	72	0-2-8	137	137	
,	111011	73	0-3-8	054	054	153
%	111100	74	0-4-8	045	045	154
=	111101	75	0-5-8	075	075	176
"	111110	76	0-6-8	042	042	177
!	111111	77	0-7-8	041	041	132

APPENDIX F

POWERS OF 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 287 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125

RDA124-38

BINARY NUMBER SYSTEM (BASE 2)

S	B +10	B +9	B +8	B +7	B +6	B +5	B +4	B +3	B +2	B +1	B +0	B -1	B -2	B -3	B -4	B -5
1	1024	512	256	128	64	32	16	8	4	2	1	.500	.250	.125	.0625	.03125

OCTAL NUMBER SYSTEM (BASE 8)

S	B +5	B +4	B +3	B +2	B +1	B +0	B -1	B -2	B -3	B -4	B -5
1	32768	4096	512	64	8	1	.125000	.015625	.001953	.0002441	.0000305
2	65536	8192	1024	128	16	2	.250000	.031250	.003906	.0004882	.0000610
3	98304	12288	1536	192	24	3	.275000	.046875	.005859	.0007324	.0000915
4	131072	16384	2048	256	32	4	.500000	.062500	.007812	.0009765	.0001220
5	163840	20480	2560	320	40	5	.625000	.078125	.009765	.0012207	.0001525
6	196608	24576	3072	384	48	6	.750000	.093750	.011718	.0014648	.0001831
7	229376	28672	3584	448	56	7	.875000	.109375	.013671	.0017089	.0002136

RDA124-39

WORKBOOK
E3ABR51131B 000
KDA-471

Technical Training

Programming Specialist (Honeywell)

COMPUTER PROGRAMMING PRINCIPLES WORKBOOK

April 1976

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler 9-0338

DO NOT USE ON THE JOB

WB 3ABR51131B
KDA-471
April 1976

COMPUTER PROGRAMMING PRINCIPLES WORKBOOK

TABLE OF CONTENTS

PART I - CLASSROOM EXERCISES

TITLE PAGE

Decimal - Binary Number Conversion 1
Decimal - Octal Number Conversion 3
Octal - Binary Number Conversion 4
Addition and Subtraction 5
Preface to Flowchart Problems 7
Sequence Flowcharts 8
Branching Flowcharts 11
Loop Flowcharts 16
Search Flowcharts 18
Sort Flowcharts 20
Insertion Flowcharts 22
Deletion Flowcharts 24
Merge Flowcharts 27
Flowchart Analysis 30
Flowchart Correction 35

PART II - HOMEWORK EXERCISES

TITLE PAGE

Decimal - Binary Number Conversion 37
Decimal - Octal Number Conversion 38
Octal - Binary Number Conversion 39
Addition and Subtraction 42
Sequence Flowcharts 43
Branching Flowcharts 49
Loop Flowcharts 57
Search Flowcharts 60
Sort Flowcharts 68
Insertion Flowcharts 79
Deletion Flowcharts 86
Merge Flowcharts 93
Flowchart Analysis 97
Flowchart Correction 108

COMPUTER PROGRAMMING PRINCIPLES WORKBOOK

OBJECTIVES

When you have completed the exercises in this workbook, you will be able to:

1. Solve mathematical problems involving both conversion of and computation with binary, octal, and decimal numbers.

2. Analyze given problems and construct flowcharts which show solutions to the given problems.

3. Analyze given flowcharts and correct any errors in them.

PROCEDURES

This workbook contains exercises to give you practice in working computer math and flowcharting problems. It is divided into two parts. Part I contains exercises for you to work during class. Your instructor will assign each of these problems after he presents the information pertaining to that particular subject. Part II contains similar problems which will be assigned as homework.

The Table of Contents groups the exercises by problem types. These titles can be used as a cross-reference to subject matter headings in the student text for assistance in solving the problems.

DECIMAL-BINARY NUMBER CONVERSION

Express the following decimal numbers in powers of 10.

1. 4832.3 ____________________

2. 28.169 ____________________

3. Complete the powers of 2 chart by inserting the appropriate decimal equivalents.

2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}

Convert the following decimal numbers to equivalent binary numbers.

4. 26_{10} ____________________$_2$

5. 37_{10} ____________________$_2$

6. 101_{10} ____________________$_2$

7. 711_{10} ____________________$_2$

8. 1024_{10} ____________________$_2$

9. $.5_{10}$ ____________________$_2$

10. $.125_{10}$ ____________________$_2$

11. $.625_{10}$ ____________________$_2$

12. $.1875_{10}$ ____________________$_2$

13. $.0625_{10}$ ____________________$_2$

14. 27.3125_{10} ____________________$_2$

15. 136.5625_{10} ____________________$_2$

16. 34.875_{10} ____________________$_2$

17. 207_{10} ____________________$_2$

18. 3052_{10} ____________________$_2$

19. $.5625_{10}$ ____________________$_2$

20. 18.75_{10} ____________________$_2$

21. 34.3125_{10} ____________________$_2$

22. 163.6875_{10} ____________________$_2$

Convert the following binary numbers to equivalent decimal numbers.

23. 1110.01_2 ____________________$_{10}$

24. 10101.101_2 ____________________$_{10}$

25. 10001.0101_2 ____________________$_{10}$

26. 11111.111_2 ____________________$_{10}$

27. 1110_2 ____________________$_{10}$

28. 110101_2 ____________________$_{10}$

29. 1010110_2 ____________________$_{10}$

30. 1011001_2 ____________________$_{10}$

DECIMAL-OCTAL NUMBER CONVERSION

Convert the following decimal numbers to equivalent octal numbers.

31. 42_{10} ______________________ $_8$

32. 285_{10} ______________________ $_8$

33. $.625_{10}$ ______________________ $_8$

34. $.328125_{10}$ ______________________ $_8$

35. 230.6875_{10} ______________________ $_8$

36. 426.78125_{10} ______________________ $_8$

37. 464_{10} ______________________ $_8$

38. $.09375_{10}$ ______________________ $_8$

39. 127.109375_{10} ______________________ $_8$

40. 633.2578125_{10} ______________________ $_8$

Convert the following octal numbers to equivalent decimal numbers.

41. 125_8 ______________________ $_{10}$

42. 1432_8 ______________________ $_{10}$

43. $.77_8$ ______________________ $_{10}$

44. $.05_8$ ______________________ $_{10}$

45. 32.52_8 ______________________ $_{10}$

46. 1572.6_8 ______________________ $_{10}$

196

47. 253_8 ____________________ $_{10}$

48. 4064_8 ____________________ $_{10}$

49. 3216_8 ____________________ $_{10}$

50. 2222_8 ____________________ $_{10}$

OCTAL-BINARY NUMBER CONVERSION

Convert the following octal numbers to equivalent binary numbers.

51. 43_8 ____________________ $_2$

52. 27_8 ____________________ $_2$

53. $.45_8$ ____________________ $_2$

54. $.33_8$ ____________________ $_2$

55. 126.61_8 ____________________ $_2$

56. 344.7_8 ____________________ $_2$

Convert the following binary numbers to equivalent octal numbers.

57. 11111_2 ____________________ $_8$

58. 101010_2 ____________________ $_8$

59. $.11101_2$ ____________________ $_8$

60. $.1001_2$ ____________________ $_8$

61. 1110.10101_2 ____________________ $_8$

62. 11101011.1_2 ____________________ $_8$

190

ADDITION AND SUBTRACTION

Add the following binary numbers, expressing the sum as a binary number.

63. 101101 + 10110	64. 110110 + 10110	65. 101011 + 10101	66. 10011110 +10110110

Find the difference between the following binary numbers, expressing your answer as a binary number.

67. 101101 - 10101	68. 10111 -10001	69. 100101 - 10110	70. 111111 - 11111

Perform the specified arithmetic operation on the following unsigned octal numbers.

71. 3731 +4720	72. 2235 +4073	73. 4046 -3665	74. 12345 - 7654

Add the following binary numbers. Use the rules for signed number addition and express each answer as a signed binary number.

75. (+1010101) +(+ 101101)	76. (+1010111) +(- 101101)	77. (-10011110) +(- 1100101)	78. (-10101) +(+11011)

Subtract the following binary numbers. Express each answer as a signed binary number.

79. (+101011) -(+101101)	80. (+101110) -(-100100)	81. (-110001) -(-101101)	82. (-10101) -(- 1110)

Subtract the following binary numbers. Express each answer as a signed binary number.

83. (+111101)
-(+100011)

84. (+100110)
-(+110011)

85. (+11010)
(-10110)

86. (+10101)
-(-10111)

87. (-101101)
-(- 10101)

88. (-1001110)
-(-1011010)

89. (-1111011)
-(+1101010)

90. (-100001)
-(+100111)

Add the following octal numbers. Express each answer as a signed octal number.

91. (-273)
+(-527)

92. (+342)
+(-721)

93. (+565)
+(+454)

94. (-362)
+(+246)

Subtract the following octal numbers. Express each answer as a signed octal number.

95. (+634)
-(+326)

96. (+654)
-(-321)

97. (-431)
-(-134)

98. (-777)
-(+333)

Subtract the following octal numbers. Express each answer as a signed octal number.

99. (+675)
-(-567)

100. (+316)
-(-631)

101. (+547)
-(+457)

102. (+206)
-(+620)

103. (-605)
-(-504)

104. (-300)
-(-542)

105. (-306)
-(+400)

106. (-472)
-(+274)

PREFACE TO FLOWCHART PROBLEMS

The flowchart problems in this workbook are designed to teach students flowcharting techniques from a general problem-solving viewpoint, not from a specific compiler language viewpoint. With a good foundation in flowcharting techniques, the student should be able to readily apply them to the language he is using (i.e., FORTRAN, GMAP, or COBOL).

The following conventions will apply to the flowchart problems in this workbook:

1. Some problems may reference files in memory, which for the purposes of this workbook will be treated as tables in core memory (similar to the way COBOL treats tabular data). The records of these files (and the fields within these records) may be accessed by referencing the file's symbolic name (or the field's symbolic name) with an integer constant or variable subscript. For example, consider the file named STUFIL with record format:

NAME	ADDR	COURSE	GD
1-20	21-45	46-69	70

Record five of STUFIL could be accessed by writing STUFIL(5). Likewise, the NAME field of record 44 could be accessed by writing NAME(44), the COURSE field of record 15 could be accessed by writing COURSE(15), or the GD field of record N could be accessed by writing GD(N), where N equals an integer constant. Remember, this method of accessing records and fields within records is only for problems in this workbook and may not work in some compiler languages (i.e., FORTRAN).

2. If temporary storage locations are used in a problem, they will be large enough to store any record in that problem.

3. The words "draw a flowchart" will be abbreviated "DAF."

4. Some problems may not include all the variables necessary for the solution of the problem or may not include record format for records of a file. In these cases the student should make up variable names and record descriptions as necessary to solve the problem.

SEQUENCE FLOWCHARTS

1. The following variables are given:

UTIL - amount of utility bills for a month.
LOAN - installment loan payment.
HOUSE - mortgage payment on house.
CAR - car payment.
GRO - amount budgeted for groceries.
MISC - amount budgeted for miscellaneous.
PAY - income for the month.

DAF to show the operations required to compute the following values:

TOTAL - total expenses for the month.
SAVE - difference between TOTAL and PAY.

2. The computer refers to the base and height of a triangle by symbolic names BASE and HGT, the length and width of a rectangle by LONG and WIDE, and the length of one side of a square by SIDE. DAF that will show the operations required to: (1) compute the area of the triangle and store the result in TRI; (2) compute the area of the rectangle and store the result in RECT; (3) compute the area of the square and store the result in SQR; (4) compute the perimeter of the rectangle and store the result in PER; and (5) compute the perimeter of the square and store the result in PERSQ.

The formulas needed to solve this problem are listed below.

Area of triangle = 1/2 (base * height)
Area of rectangle = length * width
Area of square = length ** 2
Perimeter of rectangle = 2 * length + 2 * width
Perimeter of square = 4 * length

202

3. D/F to show the operations required to accomplish the following:

(1) Read a card containing the variables LARGE and SMALL; (2) add LARGE to SMALL and store the sum into SUM; (3) subtract SMALL from LARGE and store the difference into DIFF; (4) multiply LARGE times SMALL and store the product into PROD; (5) divide LARGE by SMALL and store the quotient into QUOT, and (6) write out SUM, PROD, DIFF, and QUOT.

196

4. CIV contains the height of a potential recruit. AMN contains the minimum height required. Compare CIV to AMN to see if the civilian can be recruited. If he can be recruited, set LOC equal to 1. If he cannot be recruited, set LOC equal to 2. DAF.

5. LITE contains the status of a traffic light (1 = Red, 2 = Green). Set CAR equal to 7 if the light is green and to 6 if the light is red. DAF.

198

6. Select the proper key to open a door and store its number in SAVE. The serial numbers of the three keys are stored in KONE, KTWO, and KTHREE. The key with the largest serial number will open the door. Serial numbers are unique. DAF.

7. Check the values of BEN and ROG. If both values are zero, set SMITH equal to 0. If both values are ones, set SMITH equal to 1. If one value is zero and the other is one, set SMITH equal to 2. DAF.

8. AXE, YUM, and ZOO are integers greater than zero. If AXE equals YUM and is less than ZOO, set PEN equal to the value of YUM**5/AXE**3 and stop. If AXE is greater than YUM and equal to ZOO, set PEN equal to the value of YUM + AXE * (AXE - YUM) and stop. If AXE is less than YUM and greater than ZOO, set PEN equal to the value of AXE**2/YUM**4 and stop. For all other conditions, set PEN equal to the value of AXE + YUM + ZOO and stop. DAF.

208

LOOP FLOWCHARTS

9. Count the number of Air Force, Army, Navy, Coast Guard, and Marine personnel in a group of military being processed through an Air Terminal Processing Center. These personnel are processed one at a time, and the data pertaining to branch of service is recorded as status information as the card is punched. After 1,000 personnel have been processed, write out the title and number for each branch of service. DAF.

202

10. A manufacturer is producing 40 jeeps for military use. Some of the jeeps are to be supplied to the Air Force and some to the Army. The Air Force jeeps are to be painted blue, and the Army jeeps are to be painted olive drab. Read cards containing WHOFOR and SERNO. Check WHOFOR to see if the jeep is for the Air Force or the Army. Write out the serial number and appropriate color. When all cards have been read and checked, write out the number of jeeps processed for the Air Force and the Army.

SEARCH FLOWCHARTS

11. KARS is a file in memory with 10 records, each containing the following variables: NAME - brand of car; PRICE - car base price; ACC, AC, AUTO - prices for car optional equipment. VALUE is another variable, which indicates the maximum price we wish to pay for a car. DAF to accomplish the following:
    (1) Compare the price we wish to pay with the price of each car fully equipped.
    (2) If more than one car is found at the desired price or lower, deduct $25 from the price we wish to pay and search the file again.
    (3) If no car fits the requirements, add $25 to the price we wish to pay and search the file again.
    (4) When the desired car is found, write out the brand name and price. Only one car will meet the requirements.

NAME 1-20	PRICE 21-27	ACC 28-32	AC 33-37	AUTO 38-42

12. Describe the record format for a file of records which could contain the names, ages, and marital statuses of all your class members. Also define a variable which can be used to store the average age of your class members. DAF to compute and write out the average age of your class members.

SORT FLOWCHARTS

13. KLASS is a file in memory with 100 records, each containing the following variables: STUNR - student identification number and GRADE - class grade average. STOR is a temporary location in memory.

STUNR	GRADE
1-9	10

DAF to sort the records of KLASS into descending order based on key field GRADE.

206

14. WATER is a file in memory with 50 records, each containing the following variables: IMP - water impurities and SOURCE - source of impurities.

IMP	SOURCE
1-4	5-19

DAF to sort the records of WATER into ascending order based on key field IMP, so that the water source with the smallest amount of impurities is listed first.

INSERTION FLOWCHARTS

15. EMPLOY is a file in memory with up to 200 records, each containing the following variables: NAME - employee name; SSAN - employee social security number; and JOB - job title. NUMB is another variable which indicates the number of records in EMPLOY. NUREC is a single record to be inserted into EMPLOY, and has the following variables: NUNAME, NUSSAN, and NUJOB. DAF to insert this new employee into EMPLOY.

NAME 1-20	SSAN 21-29	JOB 30-45

NUNAME 1-20	NUSSAN 21-29	NUJOB 30-45

EMPLOY is sorted in ascending order based on Keyfield SSAN.

16. PARTS is a file in memory with up to 258 records, each containing the following variables: PNUM - part number and QTY - quantity of part. NENT is another variable which indicates the number of records in PARTS. NPARTS is a file in memory with up to 50 records, each containing the following variables: NPNUM and NQTY. LST is another variable which indicates the number of records in NPARTS. STOR is a temporary storage location in memory.

PNUM	QTY
1-5	6-8

NPNUM	NQTY
1-5	6-8

DAF to show the operations required to insert NPARTS records in PARTS. PARTS is sorted in ascending order based on key field PNUM.

DELETION FLOWCHARTS

17. TEAM is a file in memory with up to 75 records, each containing the following variables: PLAYER - football team member number and POSITION - team position. There is a stack of cards in the card reader, each containing the variable DELETE - football team member number to be cut from squad. ENDD is another variable which indicates the number of records in TEAM.

PLAYER 1-2	POSITION 3-17

DELETE 1-2	▨ 3-80

DAF to read the cards in the card reader and delete the indicated players from the football squad.

18. TEXT is a file in memory with up to 500 records, each containing the following variables: EDTN - year text was printed; SUBJ - text subject; and NROH - number of texts on hand. NUMB is another variable which indicates the number of records in TEXT. A school system needs a program to check for outdated texts. DAF to check TEXT for books printed before 1960 and write out the edition, subject, and number on hand. Then delete the record from TEXT.

EDTN	SUBJ	NROH
1-4	5-12	13-16

19. BILLS is a file in memory with up to 700 records, each containing the following variables: NAME - customer name and AMT - amount of customer's bill. NUMB is another variable which indicates the number of records in BILLS. RCVD is a file in memory with up to 150 records, each containing the names of all persons whose bills are paid in full. KNUMB is another variable which indicates the number of records in RCVD.

NAME 1-20	AMT 21-28

RCVDNAME 1-20

DAF to delete from the file BILLS, all records whose accounts are paid in full.

MERGE FLOWCHARTS

20. ELEC is a file in memory with 50 records, each containing the following variables: EMP - electrician badge number and NAME - electrician name. MECH is a file in memory with 75 records, each containing the following variables: MEMP - mechanic badge number and MNAME - mechanic name.

EMP	NAME
1-6	7-24

MEMP	MNAME
1-6	7-24

A company has its employees in two separate files. Both files are in ascending order. DAF to merge the two files into one new file using badge number as the key field, maintaining ascending order.

21. LIST is a file in memory with up to 1,000 records, each containing the following variables: ZIP - zip code; NAME - name of person; and ADRS - person's address. LNUM is another variable which indicates the number of records in LIST. OLIST is a file in memory with up to 750 records, each containing the following variables: OZIP, ONAME, and OADRS. OLNUM is another variable which indicates the number of records in OLIST.

ZIP	NAME	ADRS
1-5	6-22	23-35

OZIP	ONAME	OADRS
1-5	6-22	23-35

A company has purchased another mailing list (OLIST) and wishes to merge it with their present mailing list (LIST). Both files are in ascending order. DAF to merge the two files using zip code as the key field.

22. PARTS is a file in memory with up to 258 records, each containing the following variables: STNR - part number and QTY - quantity of part. PNUM is another variable which indicates the number of records in PARTS. NPARTS is a file in memory with 50 records, each containing the following variables: NSTNR and NQTY.

STNR 1-7	QTY 8-11

NSTNR 1-7	NQTY 8-11

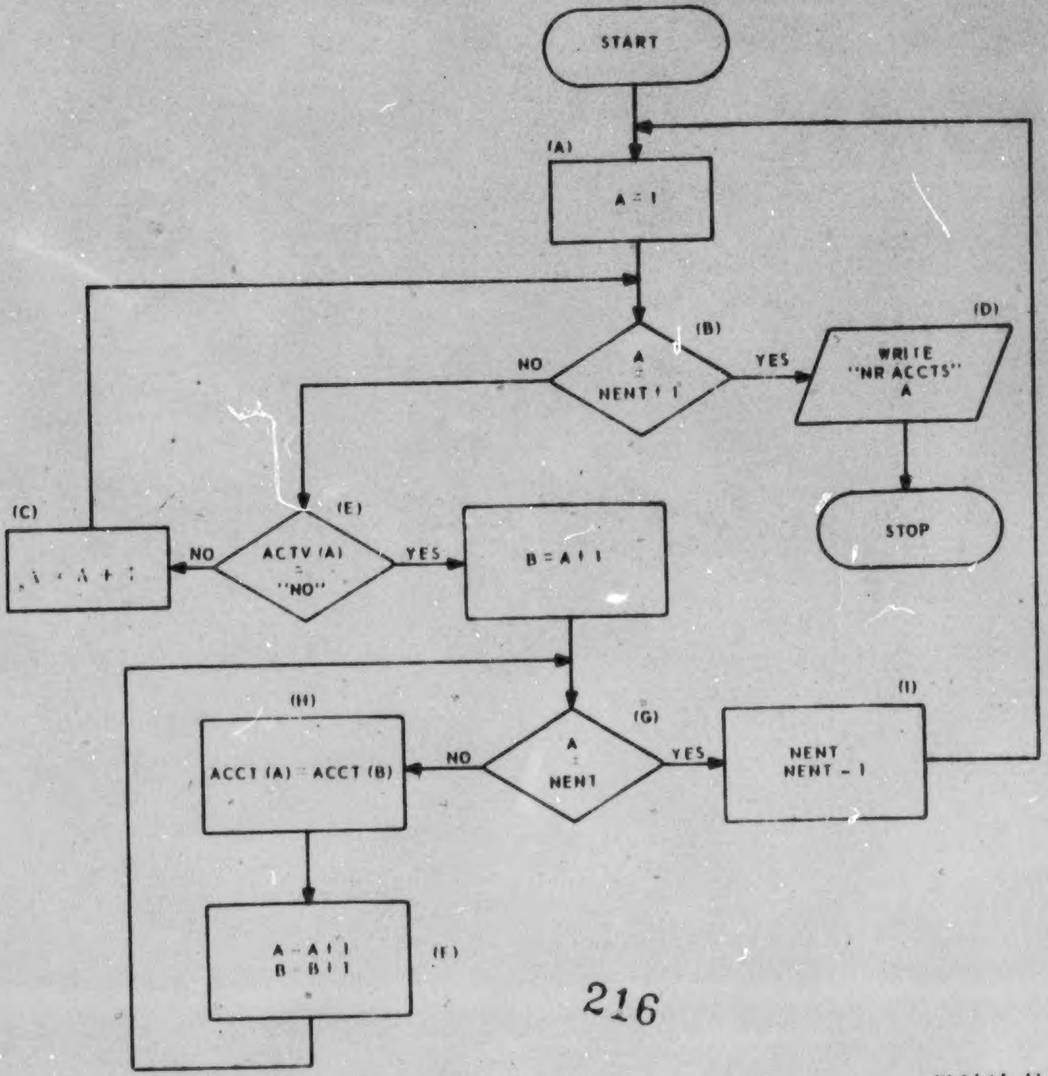
Both files are in ascending order. DAF to merge the two files into one new file, using part number as the key field. The new file is to be in descending order. If NSTNR is equal STNR, add NQTY to QTY and put one entry into the new file.

FLOWCHART ANALYSIS

For matching-type items, write the letter used to identify blocks of the flowchart into the appropriate blanks next to the descriptions of the operations. For multiple-choice items, circle the letter that identifies the correct answer.

23. ACCT is a file in memory with up to 1,000 records, each containing the following variables: BALANCE, NAME, and ACTV. NENT is another variable which indicates the number of records in ACCT.

BALANCE 1-9	NAME 10-30	ACTV 31-33



216

PPAI21-41

23. a. _____ Housekeeps NENT.

b. _____ Outputs the number of active accounts.

c. _____ Initializes an index register.

d. _____ Increments an index register.

e. _____ Increments the index registers.

f. _____ Branches out of the loop when all entries have been updated.

g. _____ Sets an entry to the value of the next sequential entry.

h. _____ What is accomplished by the flowchart for this item?

(1) Sorts the file into ascending order.

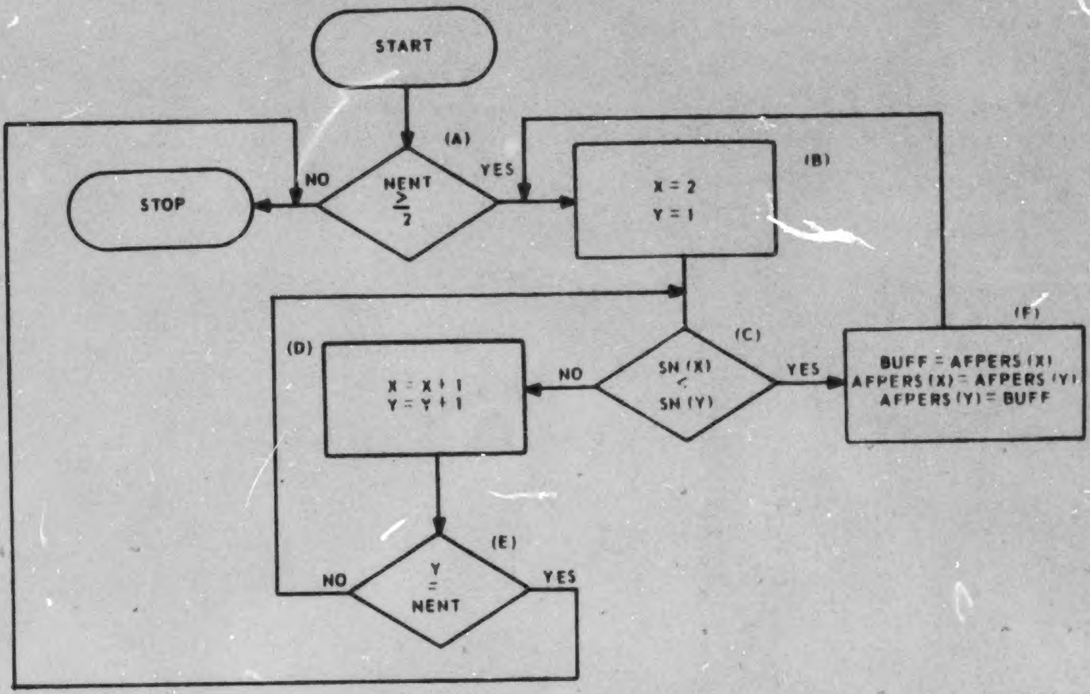
(2) Sorts the file into descending order.

(3) Inserts entries into the file.

(4) Deletes unwanted entries from the file.

24. AFPERS is a file in memory with up to 250 records, each containing the following variables: SN, NAME, and RANK. NENT is another variable which indicates the number of variables in AFPERS. BUFF is a temporary storage location in memory.

SN	NAME	RANK
1-9	10-30	31-32



RDA124-50

a. _____ Prevents entering the file if there is one or fewer entries.

b. _____ Compares the value of key items in two adjacent entries.

c. _____ Reverses the order of the values in two adjacent entries.

d. _____ Checks to see if the last entry of the file has been checked.

e. _____ Increments the index registers.

What is accomplished by the flowchart for this item?

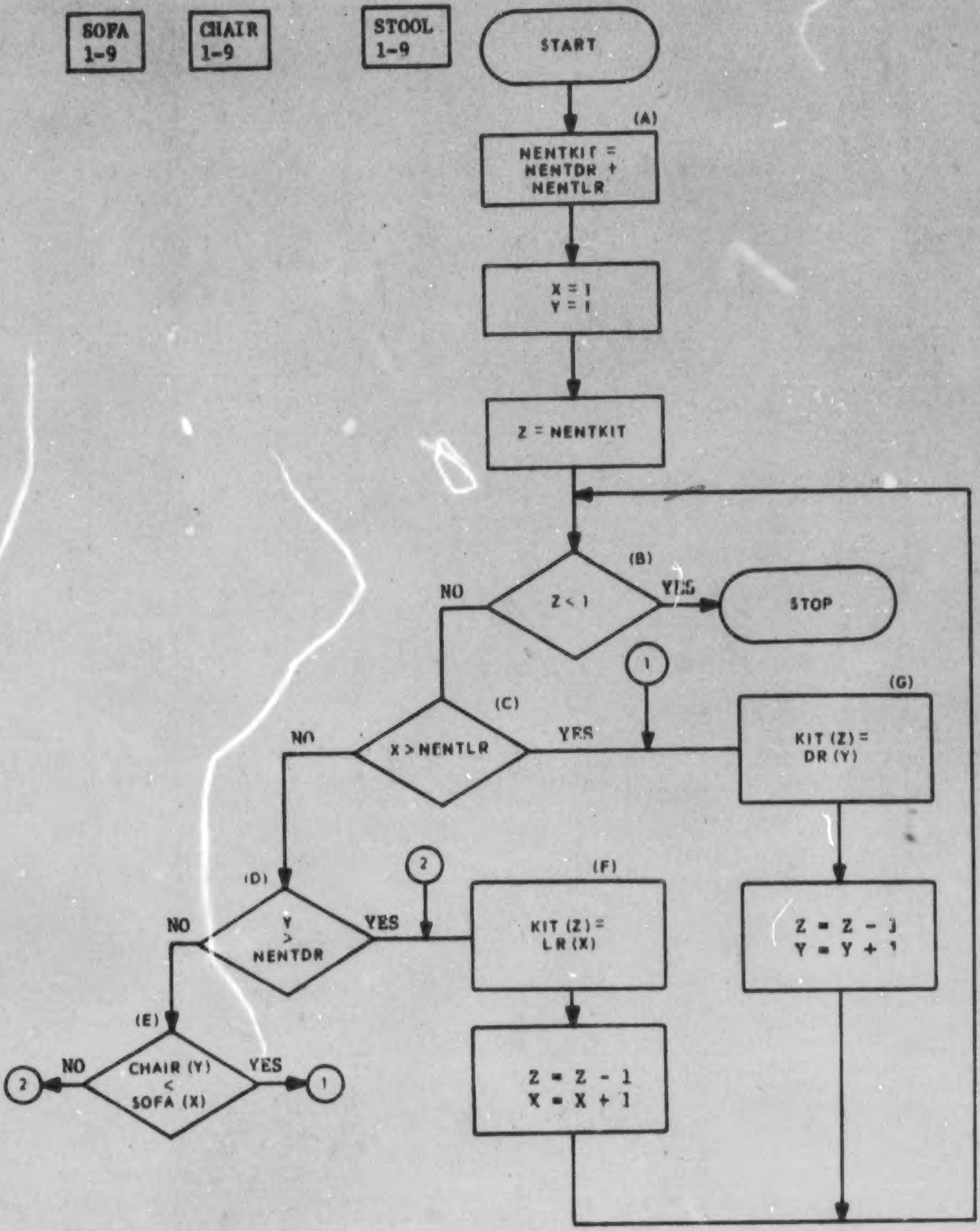
(1) Sorts the file into ascending order.

(2) Sorts the file into descending order.

(3) Inserts entries into the file.

(4) Deletes entries from the file.

25. LR is a file in memory with up to 50 records, each containing the variable SOFA. NENTLR is another variable which indicates the number of records in LR. DR is a file in memory with up to 75 records, each containing the variable CHAIR. KIT is a file in memory with up to 125 records, each containing the variable STOOL. NENTDR and NENTKIT are other variables which indicate the number of records in DR and KIT respectively.



RDA124-43

25. NOTE: Files LR and DR are in ascending order.

a. __________ Sets NENT of file KIT.

b. __________ Branches out "YES" leg to put entries from file LR into file KIT when file DR is empty.

c. __________ Branches to put the smallest value in file KIT first.

d. __________ Sets the appropriate entry into file KIT and modifies the index registers for files KIT and DR.

e. What will be the order of the file KIT after operation of the flowchart for this item?

(1) Random.

(2) Ascending.

(3) Descending.

f. What is accomplished by the flowchart for this item?

(1) Deletes entries from file KIT if they are equal to an entry found in file KIT.

(2) Inserts entries from file KIT into files LR and DR.

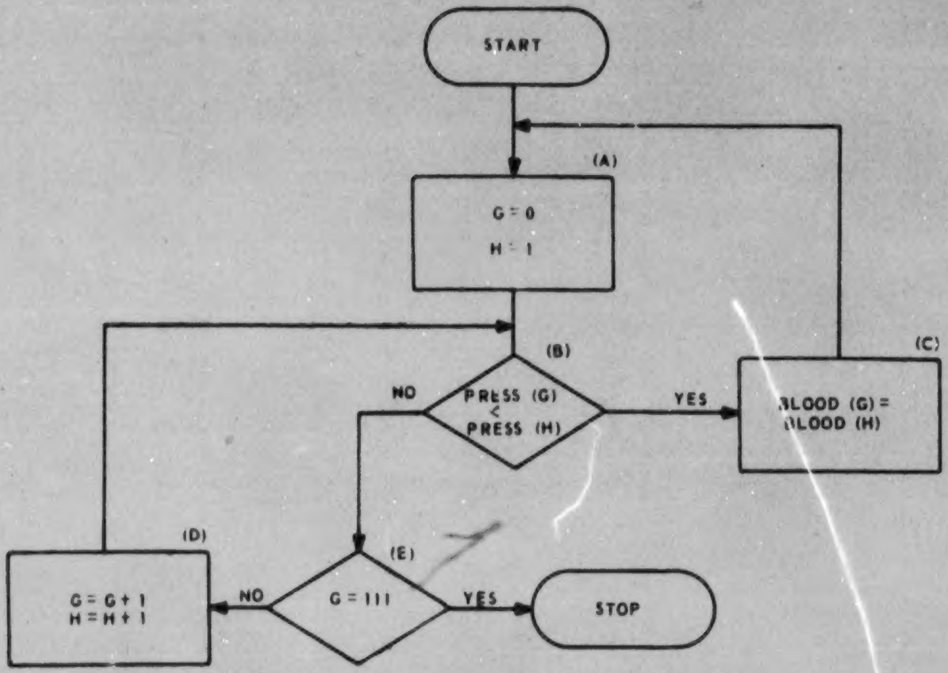
(3) Sorts files LR and DR into descending order.

(4) Merges files LR and DR into file KIT.

FLOWCHART CORRECTION

The following flowcharts contain one or more errors. Analyze the flowcharts, locate all errors, and draw corrections. If it is necessary to add a block, show its position in the flowchart.

26. BLOOD is a file in memory with up to 111 records, each containing the variable PRESS. NENT is another variable which indicates the number of records in BLOOD.

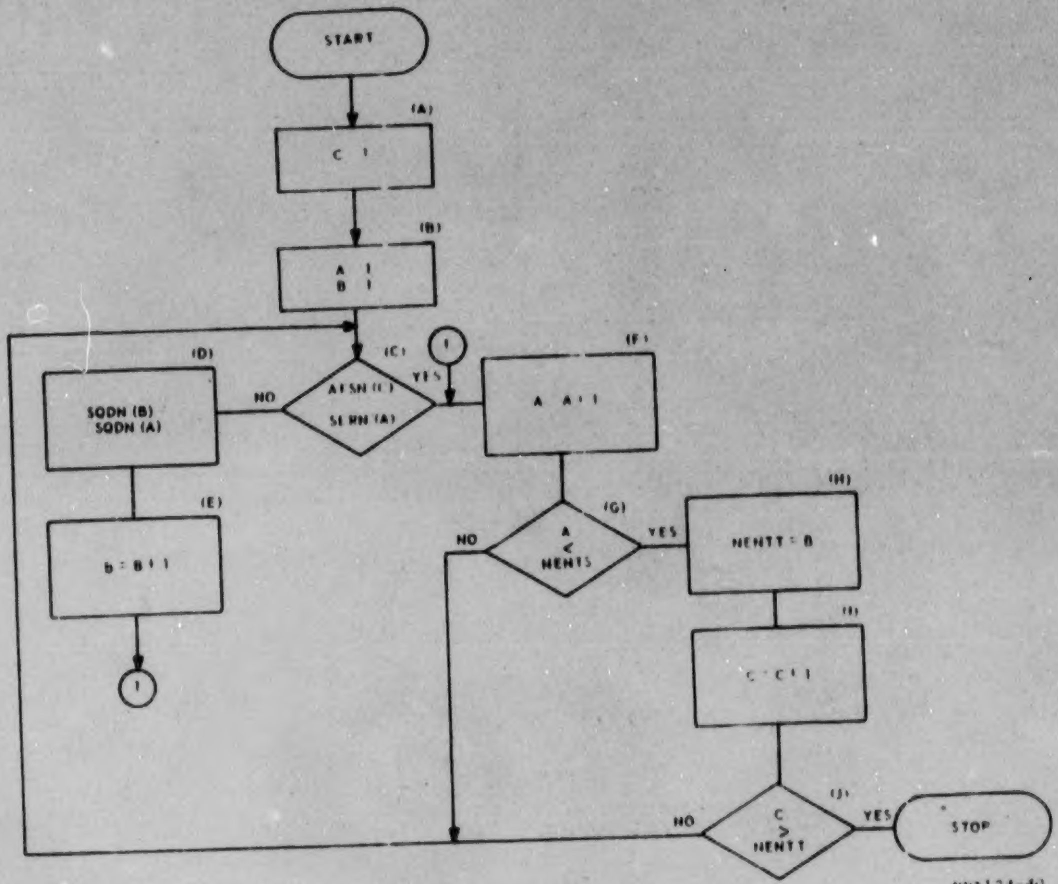


RDA124-44

MISNUMBERED - P. NO. 228 MISSING

229

27. This flowchart was designed to delete, from file SQDN, the records pertaining to personnel who have been transferred. File TRANS contains the variable SERN - serial number of transferred personnel. AFSN is the variable in SQDN records which contains the serial numbers of personnel in the squadron. NENTS and NENTT are other variables which indicate the number of records in SQDN and SERN, respectively.



RDA124-40

222

PART II

HOMEWORK EXERCISES

DECIMAL-BINARY NUMBER CONVERSION

Convert each decimal number to its equivalent binary value.

1. 35_{10} ______________________ $_{2}$

2. 61_{10} ______________________ $_{2}$

3. 105_{10} ______________________ $_{2}$

4. 327_{10} ______________________ $_{2}$

5. 226_{10} ______________________ $_{2}$

6. 0.5_{10} ______________________ $_{2}$

7. 0.125_{10} ______________________ $_{2}$

8. 0.625_{10} ______________________ $_{2}$

9. 0.1875_{10} ______________________ $_{2}$

10. 0.875_{10} ______________________ $_{2}$

Convert each binary number to its equivalent decimal value.

11. 110_{2} ______________________ $_{10}$

12. 10101_{2} ______________________ $_{10}$

13. 110101_{2} ______________________ $_{10}$

14. 1010110_{2} ______________________ $_{10}$

15. 1011001_2 ____________________ $_{10}$

16. 0.01_2 ____________________ $_{10}$

17. 0.011_2 ____________________ $_{10}$

18. 0.0101_2 ____________________ $_{10}$

19. 0.1001_2 ____________________ $_{10}$

20. 0.1101_2 ____________________ $_{10}$

DECIMAL-OCTAL NUMBER CONVERSION

Convert each decimal number to its equivalent octal value.

21. 42_{10} ____________________ $_8$

22. 85_{10} ____________________ $_8$

23. 110_{10} ____________________ $_8$

24. 231_{10} ____________________ $_8$

25. 426_{10} ____________________ $_8$

26. 0.625_{10} ____________________ $_8$

27. 0.6875_{10} ____________________ $_8$

28. 0.328125_{10} ____________________ $_8$

29. 0.18359375_{10} ____________________ $_8$

30. 0.1015625_{10} ____________________ $_8$

Convert each octal number to its equivalent decimal value.

31. 17_8 ____________________ $_{10}$

32. 54_8 ____________________ $_{10}$

33. 100_8 ____________________ $_{10}$

34. 151_8 ____________________ $_{10}$

35. 230_8 ____________________ $_{10}$

36. 0.6_8 ____________________ $_{10}$

37. 0.31_8 ____________________ $_{10}$

38. 0.43_8 ____________________ $_{10}$

39. 0.62_8 ____________________ $_{10}$

40. 0.035_8 ____________________ $_{10}$

OCTAL-BINARY NUMBER CONVERSION

Convert each octal number to its equivalent binary value.

41. 46_8 ____________________ $_2$

42. 75_8 ____________________ $_2$

43. 261_8 ____________________ $_2$

44. 0.35_8 ____________________ $_2$

45. 0.44_8 ____________________ $_2$

233

46. 0.56_8 ____________________ $_2$

47. 35.23_8 ____________________ $_2$

48. 75.436_8 ____________________ $_2$

49. 5603.72_8 ____________________ $_2$

50. 432.151_8 ____________________ $_2$

Convert each binary number to its equivalent octal value.

51. 1011011_2 ____________________ $_8$

52. 10101100_2 ____________________ $_8$

53. 11011010101_2 ____________________ $_8$

54. 0.101011_2 ____________________ $_8$

55. 0.1011010111_2 ____________________ $_8$

56. 0.11110001_2 ____________________ $_8$

57. 1010110.10110110011_2 ____________________ $_8$

58. 10101111.101110101_2 ____________________ $_8$

59. 101110.1011110101_2 ____________________ $_8$

60. 11100111.101110101101_2 ____________________ $_8$

226

ADDITION AND SUBTRACTION

Add the following binary numbers.

61. 101101
10110

62. 110110
10110

63. 101011
10101

64. 100101
10110

Subtract the following binary numbers.

65. 1010111
101101

66. 110011110
10110110

67. 101101
10101

68. 101011
11101

Add the following binary numbers. Use the rules for signed number addition and express each answer as a signed binary number.

69. (+110110)
+(+ 11111)

70. (-10001)
+(-10111)

71. (+1010110)
+(- 111111)

72. (-1111111)
+(+1111111)

Subtract the following binary numbers.

73. (+101101)
-(+ 10101)

74. (+101101)
-(+ 10110)

75. (+1110110)
-(- 110111)

76. (-101101)
-(- 11010)

77. (-11010)
-(+ 1101)

Subtract the following binary numbers. Show your work.

78. (+110011)
-(+ 10010)

79. (+1110110)
-(+1111000)

Add the following octal numbers.

80. (-1010110)
-(- 101101)

81. (+100101)
-(-110100)

82. (-110110)
-(+ 11101)

83. (+177052)
+(+436511)

84. (-534267)
+(-425037)

85. (-630275)
+(+364153)

86. (+57034)
+(- 4622)

Subtract the following octal numbers. Show your work.

87. (+143075)
-(+ 36342)

88. (+542674)
-(+731462)

89. (-356716)
-(-143522)

90. (-437562)
-(+216452)

91. (+627535)
-(-475426)

FLOWCHART PROBLEMS

See earlier "PREFACE TO FLOWCHART PROBLEMS" for the conventions which these flowcharts follow.

SEQUENCE FLOWCHARTS

1. Variables AA, BB, CC, and DD refer to values between 0 and 5. DAF to compute the sum of these values and store that sum in TOTAL and then stop.

2. Variables ONE, TWO, and THREE refer to values between 1 and 10. DAF to place the product of ONE times THREE plus the sum of ONE and TWO into ANSW and stop.

3. Variables PRICE-1, PRICE-2, PRICE-3, and PRICE-4 refer to the cost of motels for a 4-day vacation trip. DAF to obtain the total cost and the average cost for motels. Store the results in SUM and AVG.

4. The computer contains the following variables:

NAME - employee's name.
ITP - percentage to use in computing the income tax deduction.
SSP - percentage to use in computing the social security deduction.
RHP - employee's regular hourly pay.
OHP - employee's overtime hourly pay.

DAF to show the operations required to: (1) read a card containing RPH (regular pay hours) and OPH (overtime pay hours); (2) compute employee's gross pay into GRP, his income tax deduction into ITD, his social security deduction into SSD, and the amount left after deductions (take-home pay) into THP; and (3) write out the amount of the employee's gross pay, his income tax deduction, his social security deduction, and his take-home pay.

5. DAF to compute the taxable income, social security, income tax, gross pay, and net pay for an airman. His base pay, flying pay, overseas pay, hazardous duty pay, pro-pay, quarters allowance, subsistence allowance, clothing allowance, social security factor, and income tax factor are available for use. Quarters, subsistence, and clothing allowance are not taxable. If any of the items do not apply, their values will be zero. Write out the values of gross pay, net pay, income tax, and social security.

The formulas needed to solve this problem are listed below.

Gross Pay = base pay + flight pay + pro-pay + quarters + subsistence + clothing
Taxable Income = Gross pay - (quarters + subsistence + clothing)
Social Security = Taxable income * social security factor
Tax = Taxable income * income tax factor
Net Pay = Gross Pay - (Tax + Social Security)

6. DAP to find a student's average grade for each block and his course average. The average is based on a written and a performance grade for each block, with all grades having equal weight. Grades are stored in the computer as W1, P1, W2, P2, W3, and P3. Averages should be stored as BLK1, BLK2, BLK3, and AVG. Write out these averages.

7. Given the variables AA, BB, and ANS. If AA is less than BB, set ANS equal to AA + BB and stop. If AA = BB, set ANS equal to 2**AA + 2**BB and stop. If AA is greater than BB, set ANS equal to AA - BB and stop. DAF.

8. Given the variables EE, GG, and BAB. If EE = 1 and GG = 1, set BAB equal to EE**2 * GG / GG**2 * EE. If EE is greater than 30 and GG is greater than EE, set BAB equal to EE * (GG + 3). For all other conditions stop. DAF.

9. Given the variables HH, JJ, ANS1, and ANS2. If HH/JJ is greater than zero, set ANS1 equal to 4 * HH + 5 * JJ and stop. If JJ - HH is less than 99, set ANS2 equal to (1/3) * HH + 1/12 and stop. DAF.

10. Given the variables ATE, BAD, and VALU. If ATE + BAD is less than 10, set VALU equal to 1 and stop. If ATE + BAD = 10, set VALU equal to 2 and stop. If ATE + BAD is greater than 10, just stop. DAF.

238

11. Given the variables PP, QQ, RR, ANS1, ANS2, ANS3, and ERR. If PP = QQ and QQ is less than RR, set ANS1 equal to AA**2 + BB**2. If PP = QQ and QQ = RR, set ANS2 equal to 2**AA - 2**BB. If (PP is less than QQ or RR is less than QQ) and QQ = 30, set ANS3 equal to AA**2/BB**2. For all other conditions, set ERR equal to 1 and stop. D.F.

12. DAF to examine the three values stored in variables JUNE, JANE, and JEAN. Determine which is the middle value and store that value in MNUM. All values are positive (between 0 and 50) and none are equal.

13. Given the variables AA, BB, CC, and COND. AA, BB, and CC contain values greater than zero. If AA equals BB but is less than CC, set COND equal to the value of CC squared and stop. If AA equals CC but is greater than BB, set COND equal to the value of AA squared and stop. For all other conditions, set COND equal to zero and stop.

14. Given the variables A1, A2, A3, A4, and ANSW. If the sum of A1, A2, A3, and A4 is less than 100 and their product is greater than 2,000, set ANSW equal to zero and stop. If the sum is greater than 100 and the product minus the sum is less than 500, set ANSW equal to 1 and stop. For all other conditions, write out "ERROR" and stop. DAF.

250

15. DAF to read 10 cards from the card reader, each of which contains the following variables: NAME, SCORE, and PAR. After reading a card, compute the handicap and print out NAME and HDCP. The handicap is 3/4 the difference between the golfer's score and par.

16. DAP to compute the cost of lumber to build a barn. The bill of materials is contained on six different cards in the card reader. Each card contains information about one size of lumber: NR - number of boards; LGTH - length in feet; WDTH - width in inches; and THK - thickness in inches. The price is $150.00 per 1,000 board feet. A formula for computing board feet is Board Feet = width/12*thickness*length. This gives you the number of board feet in each board. Write out the total cost of lumber. Store the cost in COST and the count in CT.

17. The Air Force Academy desires to select 30 candidates whose characteristics are on cards in the card reader. Each must be from 18 to 24 years of age, have at least a 3.5 high school average, weigh from 2 to 2.5 times his or her height in inches, and be unmarried. The information about each candidate is on a card. Each card contains the following variables: NAME, AGE, AVG, TALL, WT, and WED (1 if married, 2 if unmarried). Stop when 30 have been selected or when all cards have been processed.

SEARCH FLOWCHARTS

18. TRIP is a file in memory with up to 25 records, each containing the following variables: MILES - miles to destination and DEST - destination. ENDD is another variable which refers to the number of records in TRIP.

MILES	DEST
1-4	5-21

DAF that will compare each MILES value with the value of a single variable DIST, compute the sum of those values of MILES which are greater than the value of DIST, and store the sum into SUMM.

19. A magnetic tape contains all the message headers that were received by a switching center for the last 6 hours. Each header is a separate block or entry on the tape. DAF to read the headers in and keep a count of the number of headers for each of the precedences Z, O, P, and R. There are less than 5,000 headers and the tape is terminated by a block containing the letters "ETW." Write out the counter values before stopping.

20. VALU is a file in memory with 12 records, each containing the variable NUM, a number. KOM is another variable which contains a number to be compared to. DAF to sum the numbers which are less than KON and store the sum into ARELES, sum the numbers which are equal to KON and store the sum into AREQUAL, sum the numbers which are greater than KON and store the sum into GREAT.

21. US is a file in memory with 50 records, each containing the following variables: STATE - a state of the United States; AREA - an area of a state; and POP - the population of a state.

STATE	AREA	POP
1-11	12-15	16-22

DAF to sum the number of states having a population greater than 3*10**6 and store the sum into COUNT. Write out COUNT before stopping.

22. CIGARET is a file in memory with 100 records, each containing the following variables: FILT - (1 for filter, 2 for no filter); KING - (3 for king size, 4 for not king size); and MENT - (5 for menthol, 6 for not menthol).

FILT	KING	MENT	BRAND
1	2	3	4-25

DAF to find all cigarettes which are filter tipped, king size, and non-menthol. Write out the brand name of those brands meeting these criteria.

23. TABL is a file in memory with up to 512 records, each containing the variable NUMB - a unique number. NENT is another variable which indicates the number of records in TABL. Other single variables which are used in this problem are: KONST = a constant number and ERR - error status indicator. DAF to search TABL for the value equal to KONST. If the value is found, set ENTNO equal to the number of that file record, set ERR equal to 2 (meaning no error) and stop. If the value is not found, set ERR equal to 1 (meaning no number found) and stop.

24. There are 40 numbers stored in the file FORTY. They may be positive or negative numbers. DAF to compute two totals: (1) the sum of the positive values and (2) the sum of the negative values. Store these totals in POSIT and NEGAT, respectively.

25. ACCOUNTS is a file in memory with up to 5,000 records, each containing the following variables: ACTNO - account number and PAYMENT - amount of monthly payment. TARDY is a file in memory with up to 2,000 records, each containing the variable ACCT - account number of individuals who are late in their monthly payments. NN1 and NN2 are other variables which indicate the number of records in ACCOUNTS and TARDY, respectively. Both files are in ascending order based on account number.

ACTNO	PAYMENT
1-9	10-16

ACCT
1-9

DAF to set SHORT equal to the amount of money not yet received and write SHORT out before stopping.

SORT FLOWCHARTS

26. SENIOR is a file in memory with up to 300 records, each containing the following variables: NAME - senior student name and AVERAGE - grade average. ENDD is another variable which indicates the number of records in SENIOR. DAF to sort the file into ascending order, based on key field AVERAGE.

27. There are 110 5-digit numbers (referenced by the variable NUMS) stored in a file called NOS. DAF to sort these numbers in descending order.

28. DECK is a file in memory which contains positive non-duplicate numbers in random sequence. DAF to sort DECK into descending order, based on key field KEYCRD. There are 586 records in DECK.

29. Positive numbers are stored in random sequence in a file called POS. DAF to sort these numbers in ascending order based on key field NUMB.

30. BAT is a file in memory with up to 500 records, each containing the following variables: ROB, RAY, and ALPHA.

ROB	RAY	ALPHA
1-5	6-10	11-18

DAF to sort BAT into descending order, based on key field ROB. After BAT has been sorted, call in subroutine PYTHAGORUS'S THEOREM. Before using the subroutine, set SIDE1 equal to ROB and SIDE2 equal to RAY. The results of the computation will be placed in SIDE3. Store this result into ALPHA. Cycle through the entire file, performing this computation for each record.

31. POPEXP is a file in memory with up to 1,000 records, each containing the following variables: CITY - city; STATE - state; and POP - population of city. NENT is another variable which indicates the number of records in POPEXP.

CITY	STATE	POP
1-15	16-25	26-33

DAF to sort POPEXP in ascending order based on key field POP. When the sort is finished, set PCITY = CITY, PSTATE = STATE; and PPOP = POP and call in subroutine LPRINT to print each entry. Write out the entire table.

32. There are 100 cards in the card reader. Each card has the name and population of an American city. DAF to read these cards into a file CITY. Check each record and if the population is 500,000 or more, set SIZE equal to "LARGE" for that record. If the population is greater than 50,000 but less than 500,000, set SIZE equal to "MED." If the population is 50,000 or less, set SIZE equal to "SMALL." When all cities have been checked, sort the file into descending order based on population.

33. INPUT is a file in memory which contains up to 100 series card message headers for a given card terminal which constitutes its message traffic for one Julian day. DAF to sort the random headers into ascending sequence, based on file time in headers. Describe record format as necessary to solve this problem.

34. TBL is a file in memory with up to 1,000 records, each containing the following variables: KEE and INTEGER. NENT is another variable which indicates the number of records in TBL.

KEE	INTEGER
1-3	4-10

TBL is in random order. If SOSO equals any item KEE, sort TBL into descending order. If none are equal, stop. KEE is the key field. DAF.

35. PRES is a file in memory with up to 50 records, each containing the following variables: STATE, YEAR, and PARTY (1 for Democrat, 2 for Republican, and 3 for other). NENT is another variable which indicates the number of records in PRES.

STATE	YEAR	PARTY
1-8	9-12	13

PRES is in random order. Sort the file into ascending order based on key field YEAR, then determine if any record has the year 1838 and PARTY equals Democrat. If so, write out the state which that president came from. If this condition does not exist, write out error message. The year 1838 and Democratic party may exist more than once.

271

36. BELO is a file in memory with up to 100 records, each containing the following variables: DIGIT - number and DECRE - another number. ENDD is another variable which indicates the number of records in BELO. BELO is in random order.

DIGIT	DECRE
1-4	5-6

DAF to sort the file into ascending order, based on key field DIGIT. If duplicate numbers exist in digit, use DECRE as the secondary key field.

264

37. DATA is a file in memory with up to 365 records, each containing the variable INFO. NENT is another variable which indicates the number of records in DATA. DATA is in descending order. DAF to insert the value FACT into the file DATA in the proper location. If there is not sufficient space for a new entry, write out an error message.

38. BOWL is a file in memory with up to 88 records, each containing the following variables: NAME and HDCAP - bowler's handicap. NENT is another variable which indicates the number of records in BOWL. DAF to read ten cards containing the variables NNAME and NHDCAP, one at a time. Insert these records into the proper location of BOWL. BOWL is in ascending order based on key field HDCAP.

39. ANUM is a file in memory with up to 100 records, each containing the variable DIG - a number. ENDA is another variable which indicates the number of records in ANUM. BNUM is a file in memory with up to 50 records, each containing the variable VAL - a number. ENDB is another variable which indicates the number of records in BNUM. Both files are in ascending order, DAF to insert all the records of BNUM into their proper location in ANUM.

275

40. RINGO is a file in memory with up to 61 records, each containing the following variables: HIP and SHOW. NENT is another variable which indicates the number of records in RINGO. DAF to insert a single record containing the variables NHIP and NSHOW into the proper location in RINGO. RINGO is in descending order based on key field HIP.

268

41. ERN is a file in memory with up to 150 records, each containing the following variables: PEA and PICKER. ENDD is another variable which indicates the number of records in ERN. DAF to insert a single record containing the variables COTTON and PICK into the proper location of ERN. ERN is in descending order based on key field PEA.

42. BIB is a file in memory with up to 200 records, each containing the variable JIM. ENDA is another variable which indicates the number of records in BIB. RAY is a file in memory with up to 100 records, each containing the variable BOB. ENDB is another variable which indicates the number of records in RAY. Both files are in ascending order. DAF to insert all of the records of RAY into their proper location in BIB. If there is not enough space in BIB, write out an error message.

43. TBLO is a file in memory with up to 1,000 records, each containing the following variables: ITM and BTM. LAST is another variable which indicates the number of records in TBLO. DAF to sort TBLO into descending order based on key field ITM. Also insert a single record containing the variables NTAB and BTAB into the appropriate slot of TBLO. If TBLO is full, complete the sort and go to stop.

279

DELETION FLOWCHARTS

44. TBL is a file in memory with up to 1,000 records, each containing the following variables: ITM and BTM. NENT is another variable which indicates the number of records in TBL. DAF to sort TBL into ascending order based on key field ITM, then delete the last two records of TBL.

272

45. ONE is a file in memory with up to 237 records, each containing the variable TWO. NENT is another variable which indicates the number of records in ONE. ONE is in random order. DAF to delete from ONE all the records with a value of 17 or 46 for TWO.

46. RESULTS is a file in memory with up to 10,000 records, each containing the following variables: SERIAL - serial number, AGE, and IQ. NENT is another variable which indicates the number of records in RESULTS. RESULTS contains the serial number, age, and IQ for up to 10,000 armed service personnel. WAP to delete the records with an IQ of less than 90 and repack the file.

47. WAGES is a file in memory with up to 8,000 records, each containing the following variables: EMPNO - employee number and SALARY. ENDA is another variable which indicates the number of records in WAGES. RETIRED is a file in memory with up to 50 records, each containing the variable NUM - employee number of retired employees. ENDB is another variable which indicates the number of records in WAGES. DAF to delete all retired personnel from WAGES.

48. SUNK is a file in memory with up to 1,000 records, each containing the variable SWIM. NENT is another variable which indicates the number of records in SUNK. SUNK is in random order. DAF to delete all records of SUNK that are equal to the value of NVAL. Note all numbers are not unique.

49. THINK is a file in memory with up to 100 records, each containing the variable VAL. NENT is another variable which indicates the number of records in THINK. THINK is in random order with duplicate numbers. DAF to sort THINK into descending order. When finished, delete each entry that has a value equal to the single variable KAY.

50. SUMM is a file in memory with up to 200 records, each containing the variable XRAY. NENT is another variable which indicates the number of records in SUMM. SUMM is in random sequence. DAF to delete all entries where XRAY has a value between 100 and 150, inclusive.

278

MERGE FLOWCHARTS

51. Given variable length files UNO and DOS. The key field for UNO is EIN. The key field for DOS is AWEI. Both files are in descending order. Each has positive and duplicate numbers. DAF to merge UNO and DOS into a third file TRES, which will be in ascending order. Describe files as necessary to solve the problem.

52. PERS is a file in memory with up to 8,000 records, each containing the following variables: SERIAL - social security number; RANK - rank of individual; and NAME. NENT is another variable which indicates the number of records in PERS. PERSX is a file in memory with up to 50 records, each containing the following variables: SERIALX, RANKX, and NAMEX. NENTX is another variable which indicates the number of records in PERSX. Both files are in ascending order with SERIAL and SERIALX as the key fields. DAF to merge the two files, maintaining ascending order.

SERIAL	RANK	NAME
1-9	10-11	12-32

SERIALX	RANKX	NAMEX
1-9	10-11	12-32

53. CUSTOMER is a file in memory with up to 1,000 records, each containing variables ACCT and NAME. NENTA is another variable which indicates the number of records in CUSTOMER. NUCUST is a file in memory with up to 1,000 records, each containing variables NUACCT and NUNAME. NENTB is another variable which indicates the number of records in NUCUST.

ACCT	NAME
1-5	6-26

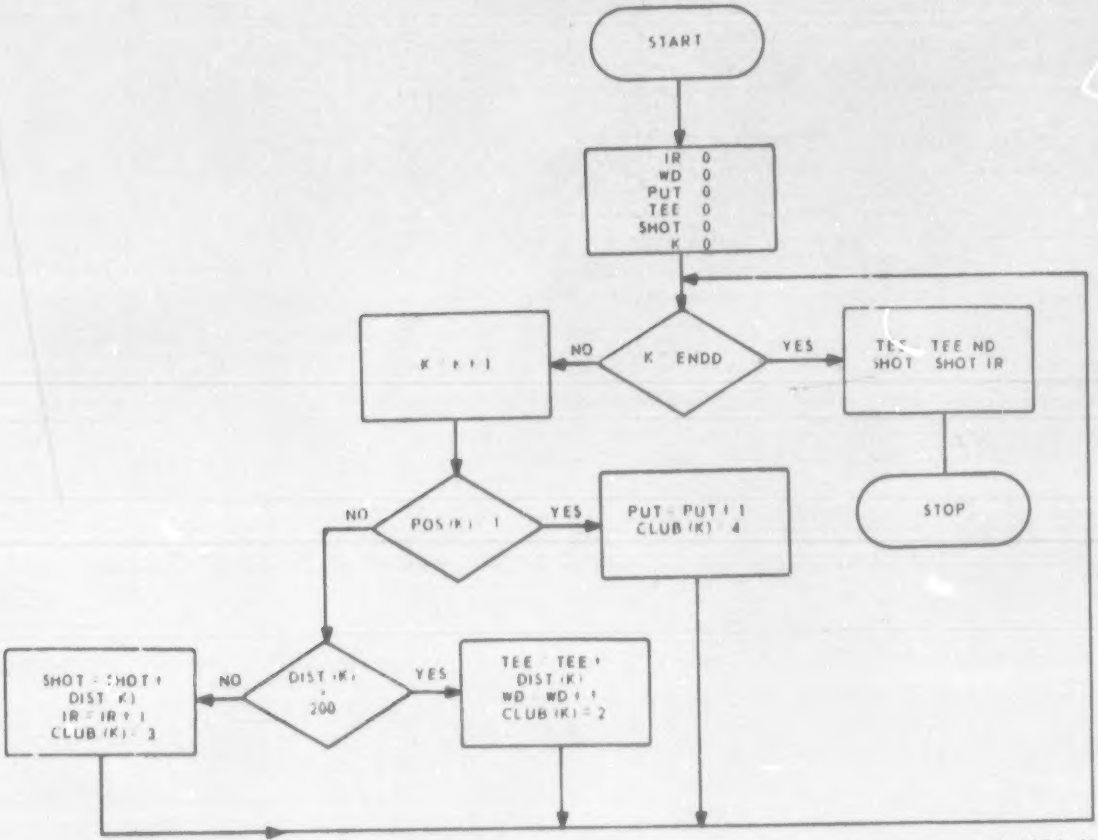
NUACCT	NUNAME
1-5	6-26

CUSTOMER is in ascending order and NUCUST is in descending order. DAF to merge the two files in ascending order. ACCT and NUACCT are the key fields. All values are unique.

54. FLOT is a file in memory with up to 100 records, each containing the following variables: TYPEF - type of ship; NUMF - number of ship; RANGE - range of ship; and SPEED - speed of ship. ENDD is another variable which indicates the number of records in FLOT. PAT is another file in memory with 10 records, each containing the following variables: TYPEP, NUMP, RANGP, and SPEEDP. FLOT is in ascending order and contains information on the ships in a Navy unit, ordered by number. PAT is in descending order and contains information on new ships being assigned, ordered by number. Merge the two files so the resulting file is in ascending order.

FLOWCHART ANALYSIS

55. GOLF is a file in memory with up to 150 records, each containing the following variables: DIST, POS (0 for fairway, 1 for green), and CLUB (2 for wood, 3 for iron, and 4 for putter). ENDD is another variable which indicates the number of records in GOLF.



RDA124-42

291

55. a. If DIST in entry number 35 is 180, what items will be set during that pass?

(1) CLUB (35), NENT (GOLF).

(2) IR, SHOT, CLUB (35).

(3) IR, SHOT, DIST (35).

(4) DIST (35), POS (35), CLUB (35).

b. Suppose this file has 80 entries. What will be the final count in K?

(1) Indeterminate.

(2) 79.

(3) 80.

(4) 150.

c. When is item CLUB set to WOOD?

(1) WD is set to 0.

(2) Distance of shot greater than 200 feet.

(3) Distance of shot equal to or less than 200 feet.

(4) Ball positioned on fairway.

d. What information is contained in item PUT when the flow is complete?

(1) Number of strokes made on green.

(2) Number of strokes hit less than 200 feet.

(3) Average distance covered by putts.

(4) Total distance covered by putts.

e. What is the maximum value item SHOT might possibly contain in any time during this flow?

(1) 150.

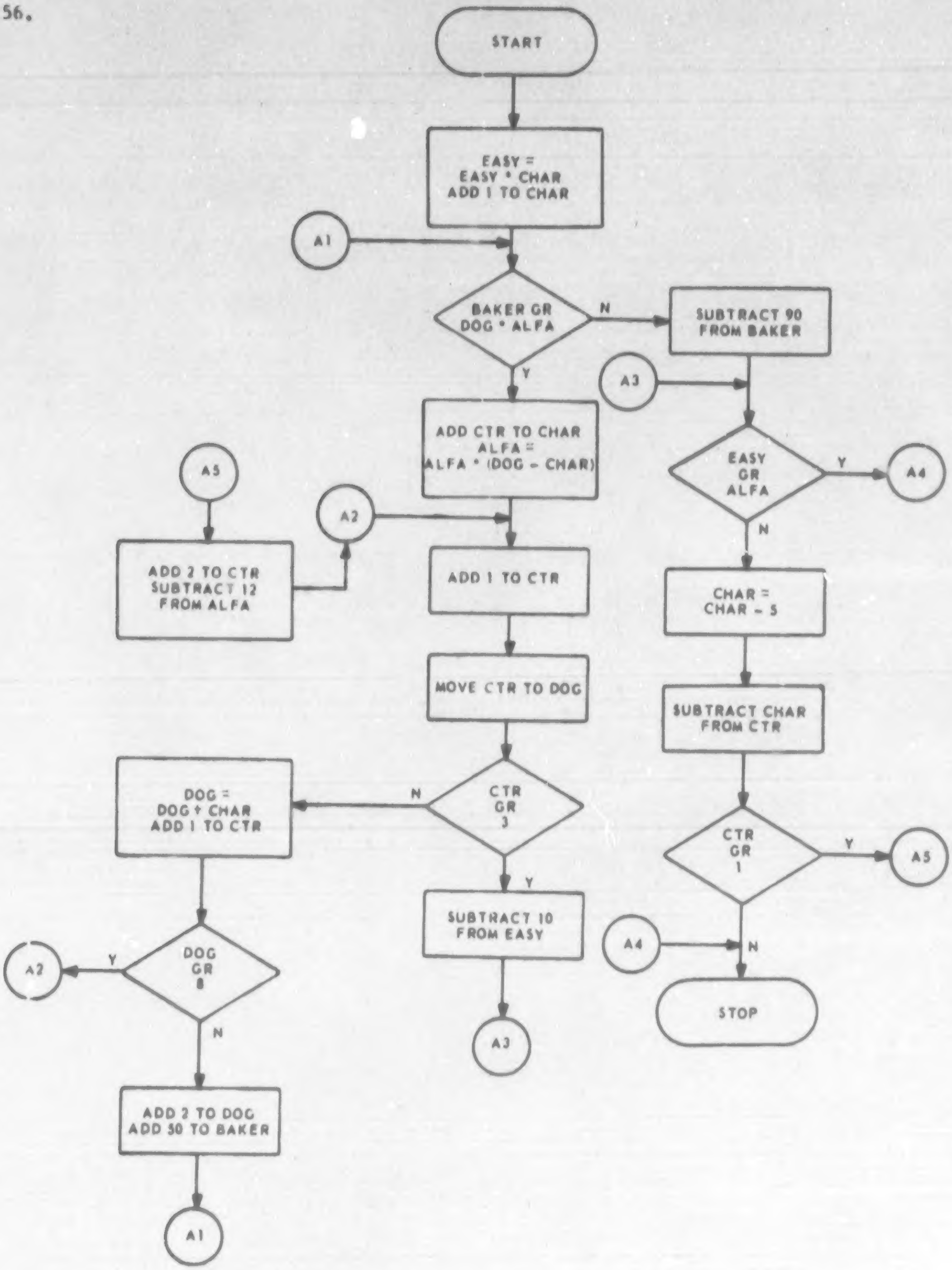
(2) 199.

(3) 29850.

(4) 30000.

284

56.


START
EASY =
EASY * CHAR
ADD 1 TO CHAR
A1
BAKER GR
DOG * ALFA
N
SUBTRACT 90
FROM BAKER
A3
Y
ADD CTR TO CHAR
ALFA =
ALFA * (DOG - CHAR)
EASY
GR
ALFA
Y
A4
A5
A2
N
ADD 2 TO CTR
SUBTRACT 12
FROM ALFA
ADD 1 TO CTR
CHAR =
CHAR - 5
MOVE CTR TO DOG
SUBTRACT CHAR
FROM CTR
DOG =
DOG + CHAR
ADD 1 TO CTR
N
CTR
GR
3
CTR
GR
1
Y
A5
Y
SUBTRACT 10
FROM EASY
A4
N
A2
Y
DOG
GR
8
STOP
A3
N
ADD 2 TO DOG
ADD 50 TO BAKER
A1

RDA124-56

293

56. List the value contained in the following variables when:

		CTR = 2	CTR = 3	CTR = 4	CTR = 5	The program STOPS
a.	ALFA					
b.	BAKER					
c.	CHAR					
d.	DOG					
e.	EASY					
	CTR	2	3	4	5	

Initial Values:

ALFA = 15

BAKER = 76

CHAR = 2

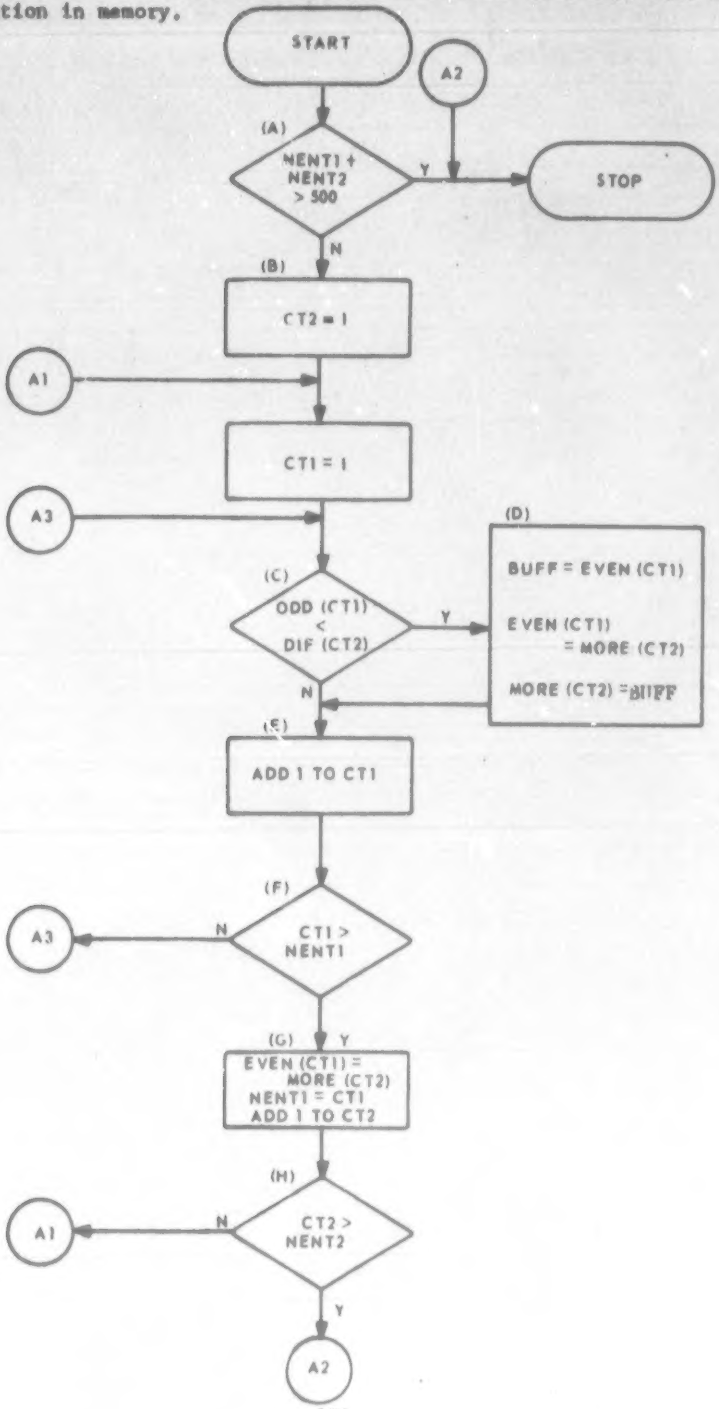
DOG = 5

EASY = 12

CTR

296

57. EVEN is a file in memory with up to 500 records, each containing the variable ODD. NENT1 is another variable which indicates the number of records in EVEN. MORE is a file in memory with up to 50 records, each containing the variable DIFF. NENT2 is another variable which indicates the number of records in MORE. BUFF is a temporary storage location in memory.



57. a. Which block insures that NENT1 contains the number of entries in file EVEN?

(1) (A)

(2) (F)

(3) (G)

(4) (H)

b. Block (A) insures that

(1) file EVEN is not sorted if there are more than 500 entries.

(2) no entries are deleted if there are less than 501 entries.

(3) the files are not merged if there are less than 501 entries.

(4) no entries are inserted into file EVEN unless there is room for all.

c. What is accomplished by the flowchart?

(1) Sorts file EVEN in ascending order.

(2) Inserts file MORE into file EVEN.

(3) Merges files MORE and EVEN.

(4) Deletes all entries from file EVEN that contain a value less than the entry in file MORE.

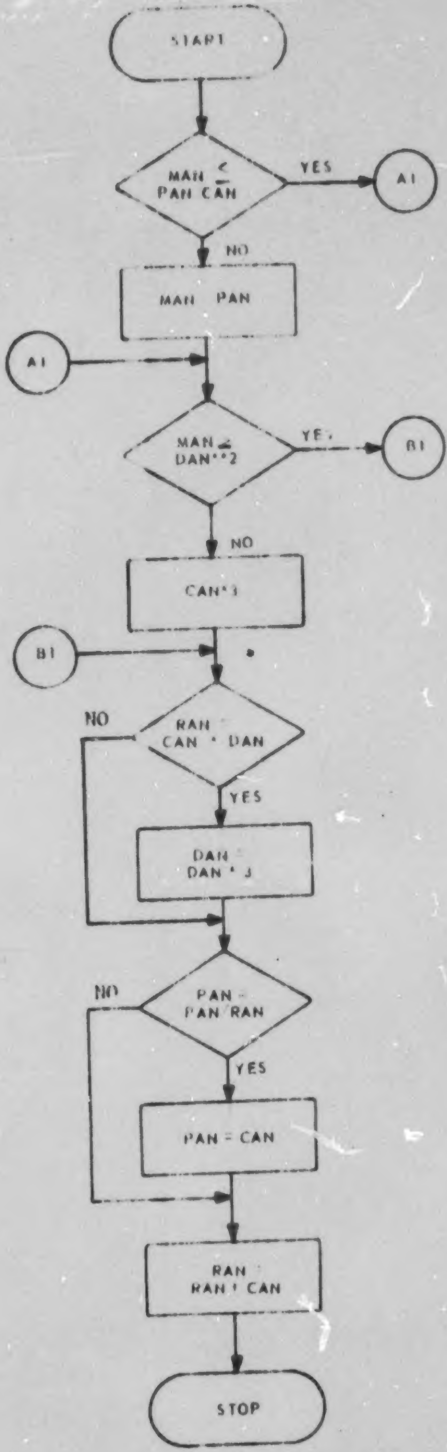
d. What is the order of file EVEN at the end of the flow, assuming that it was ordered to start with?

(1) Ascending.

(2) Descending.

(3) Random.

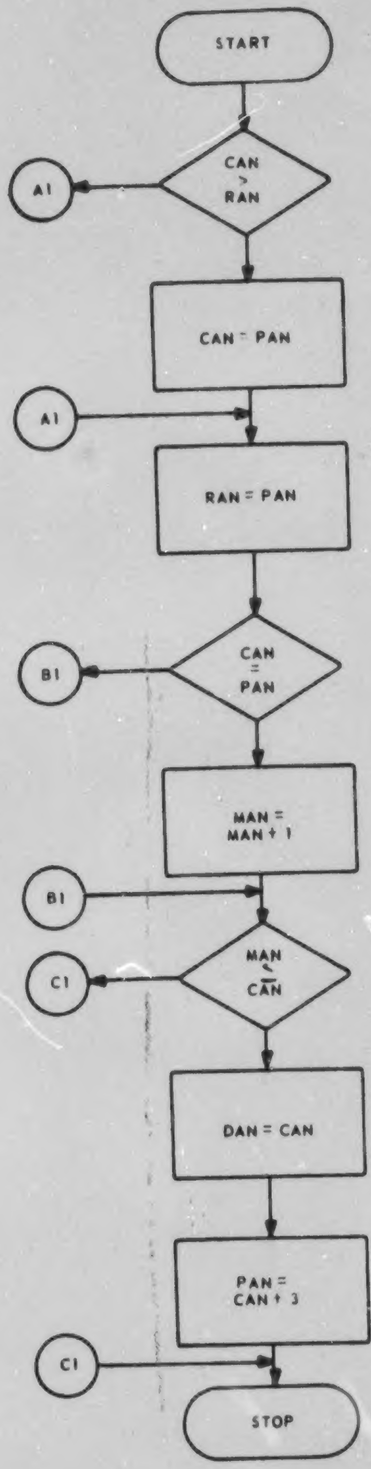
58.


START
MAN ≤ PAN CAN
YES
A1
NO
MAN PAN
A1
MAN ≤ DAN**2
YES
B1
NO
CAN*3
B1
NO
RAN = CAN * DAN
YES
DAN = DAN * 3
NO
PAN = PAN/RAN
YES
PAN = CAN
RAN = RAN + CAN
STOP

RDA124-51

58. Analyze the flowchart on the preceding page and determine the final values of A, B, C, and D for each variable using the initial values given.

	VARIABLE	INITIAL VALUE				FINAL VALUE			
		A	B	C	D	A	B	C	D
a.	RAN	0	5	11	19	___	___	___	___
b.	CAN	1	7	3	22	___	___	___	___
c.	DAN	2	9	8	6	___	___	___	___
d.	MAN	3	15	0	27	___	___	___	___
e.	PAN	4	6	1	13	___	___	___	___


START
CAN > RAN
A1
CAN = PAN
A1
RAN = PAN
B1
CAN = PAN
MAN = MAN + 1
B1
MAN ≥ CAN
C1
DAN = CAN
PAN = CAN + 3
C1
STOP

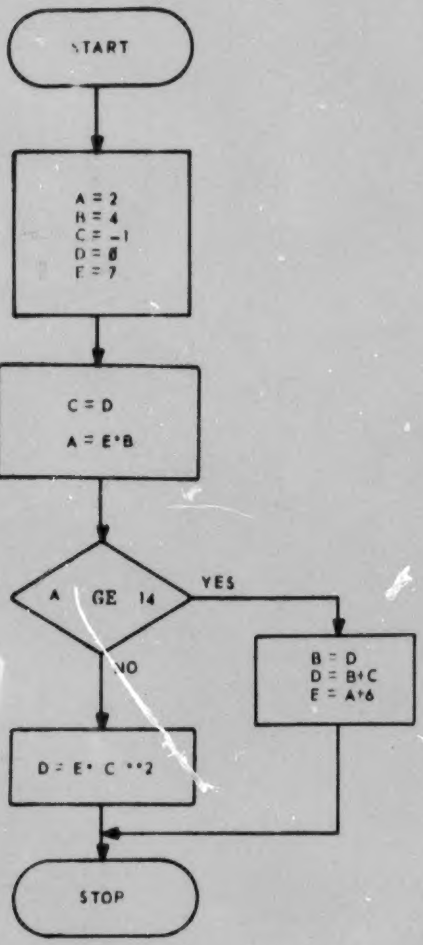
RDA124-49

59. Analyze the flowchart on the preceding page and determine the final values of A, B, C, and D for each variable using the initial values given.

	VARIABLE	INITIAL VALUE				FINAL VALUE			
		A	B	C	D	A	B	C	D
a.	RAN	0	5	11	19	___	___	___	___
b.	CAN	1	7	3	22	___	___	___	___
c.	DAN	2	9	8	6	___	___	___	___
d.	MAN	3	15	0	27	___	___	___	___
e.	PAN	4	6	1	13	___	___	___	___

60. Determine the contents of each of the following variables when "stop" is executed:

VARIABLES		FINAL VALUES
A	=	________
B	=	________
C	=	________
D	=	________
E	=	________

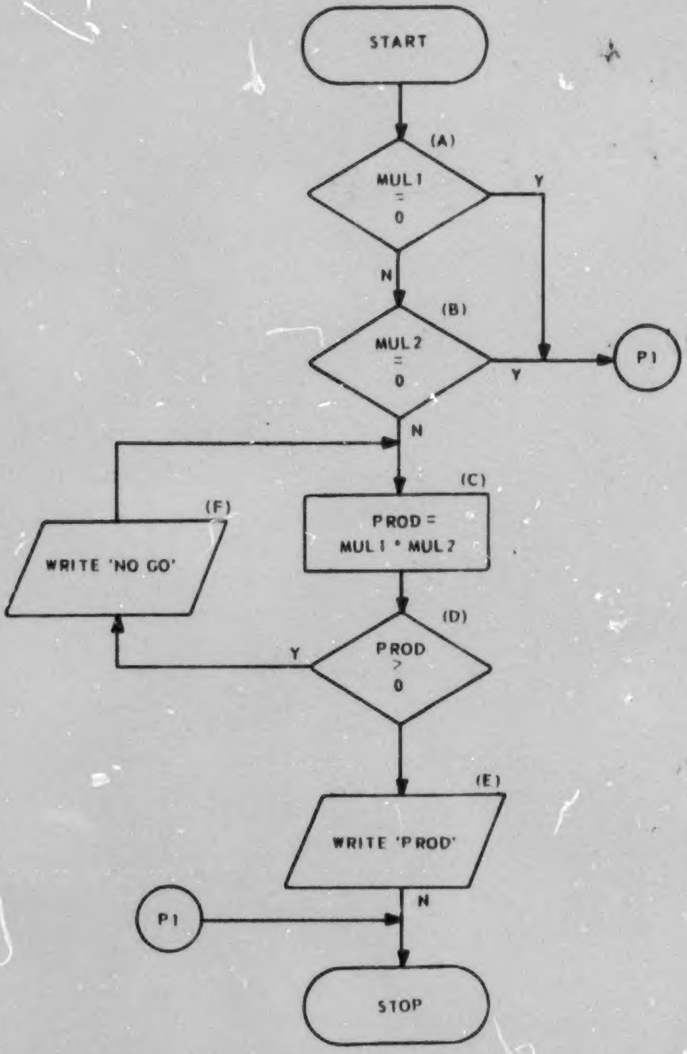


RDA124-53

301

FLOWCHART CORRECTION

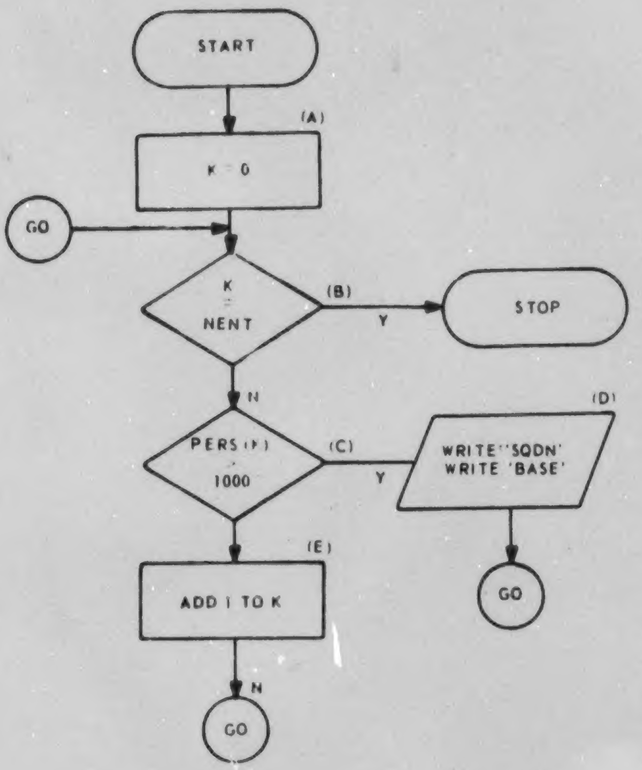
61. Correct the errors in the following flow. This flow should multiply items MUL1 and MUL2, print out the product, and, if the product is zero, print out "NO GO."



RDA124-46

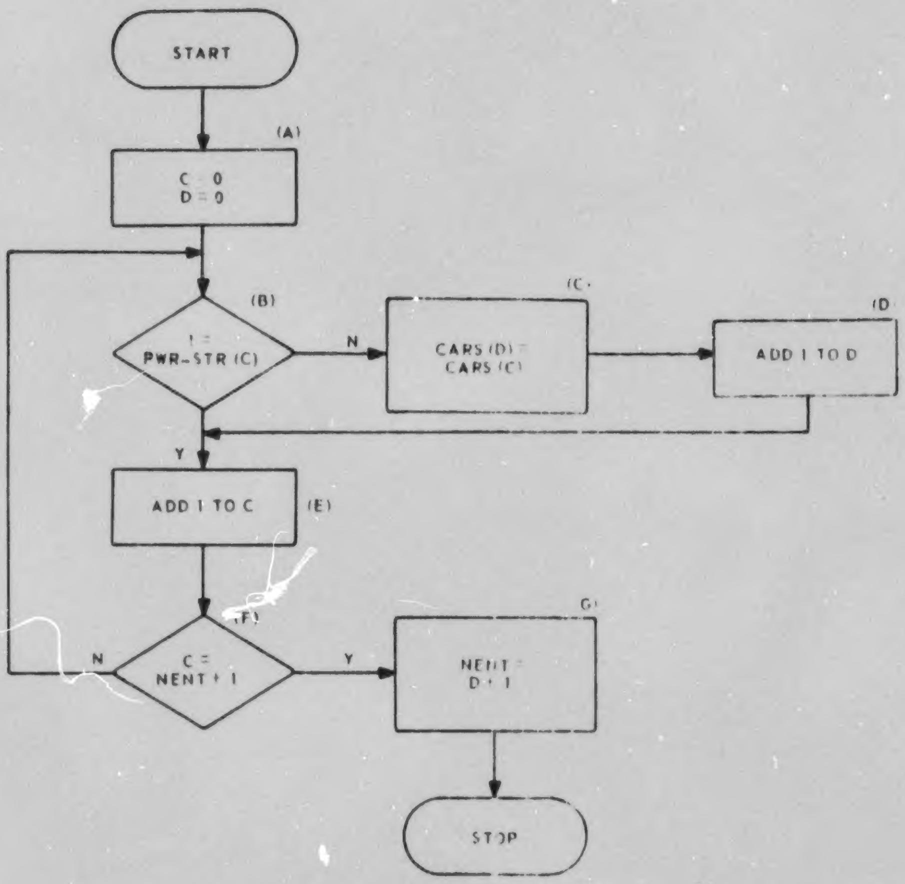
294

62. Correct the error(s) in the following flow. AF is a file in memory with up to 4,000 records, each containing the following variables: SQDN, PERS, BASE, and LOC. NENT is another variable which indicates the number of records in AF. The flow should print out the squadron and base of all squadrons having more than 1,000 personnel assigned. AF is in random order.



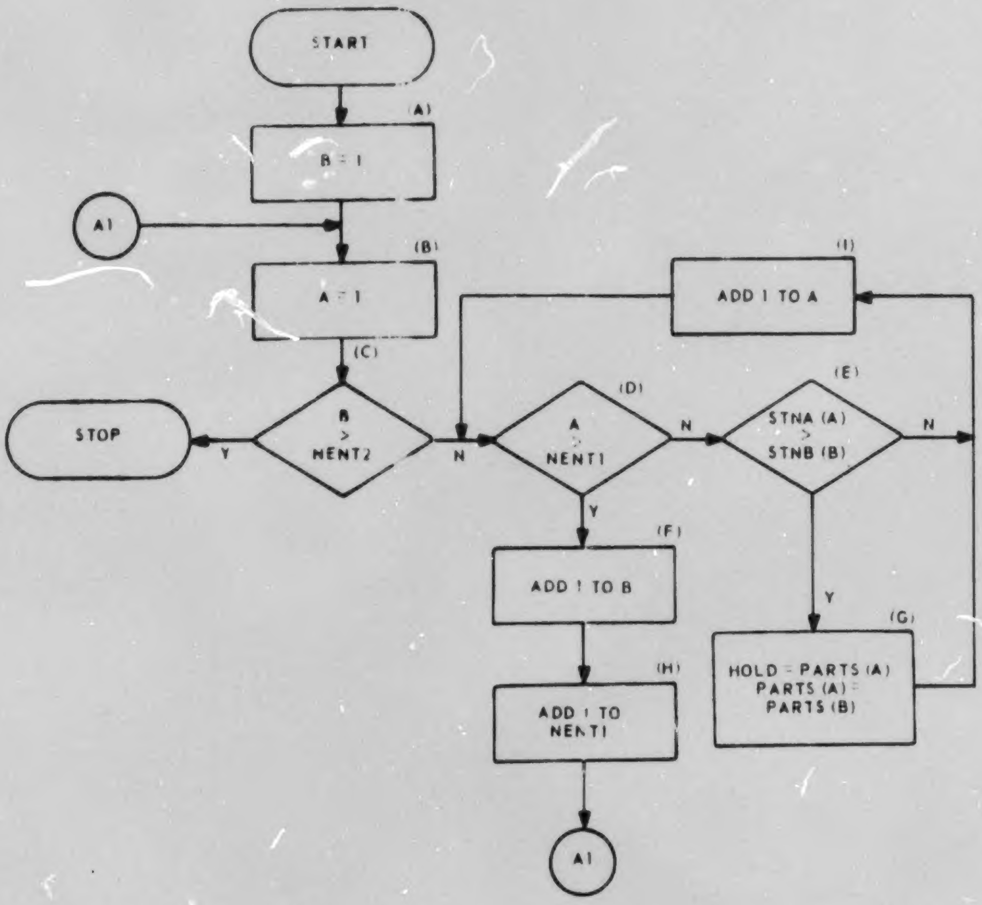
RDA124-52

63. CARS is a file in memory with up to 300 records, each containing the following variables: COST and PWRSTR (1 for yes, 2 for no). NENT is another variable which indicates the number of records in CARS. The flow should delete the records pertaining to all cars that do not have power steering (PWRSTR).



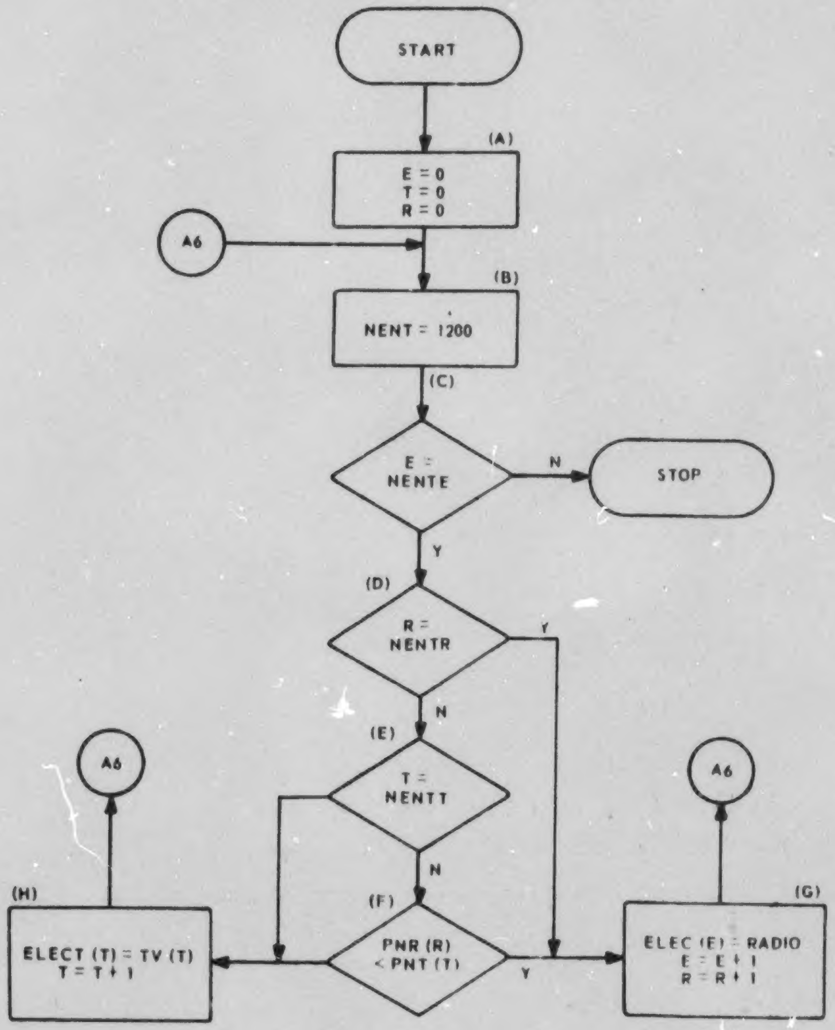
FDA124-45

64. PARTSA is a file in memory with up to 258 records, each containing the following variables: STNA and QTYA. NENT1 is another variable which indicates the number of records in PARTSA. PARTSB is a file in memory with up to 50 records, each containing the following variables: STNB and QTYB. NENT2 is another variable which indicates the number of records in PARTSB. The flow should insert PARTSB records into PARTSA. The key fields are STNA and STNB. PARTSA is in ascending order and PARTSB is in random order. When finished, PARTSA should still be in ascending order.



RDA124-48

65. RADIO is a file in memory with up to 500 records, each containing the following variables: PNR and RQTY. NENTR is another variable which indicates the number of records in RADIO. TV is a file in memory with up to 700 records, each containing the following variables: PNT and TQTY. NENTT is another variable which indicates the number of records in TV. ELEC is a file in memory with up to 1,200 records, each containing the following variables: PNE and EQTY. NENTE is another variable which indicates the number of records in ELEC. The flow should merge RADIO and TV into ELEC. The original files were in ascending order, based on PNT and PNR. The new file should also be in ascending order, based on PNE.



RDA124-47

306

PROGRAMMED TEXT
E30ZR3024D 000
KDA-305

Technical Training

Communications Computer Programmer

COMPUTER LOGIC FUNCTIONS

August 1974

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler [illegible]

DO NOT USE ON THE JOB

COMPUTER LOGIC FUNCTIONS

CONTENTS

TITLE	PAGE
Foreword	i
Contents	ii
Instructions to the Student	iii
Truth Table	iv
AND-Logic Function	1
OR-Logic Function	11
NOT-Logic Function	21
EXCLUSIVE OR-Logic Function	37

OBJECTIVES

Given a series of problems and a Truth Table containing the required combinations of AND, OR, and NOT logic functions, name the logic function and determine the result of performing the logic functions on five-digit binary values. 70% of the answers must be correct.

308

INSTRUCTIONS TO THE STUDENT

1. This programmed text is designed to present all the information and practice you need in this course in relation to Computer (Boolean) Logic Functions. It is designed for you to complete during home study but could be used during class time if desired.

2. This programmed text is designed to serve as a review for persons who have previously studied Computer Logic Functions or as a complete lesson for persons who are just beginning to study Computer Programming Principles.

3. Specific instructions are presented as you progress through this text. You must follow these instructions in order to gain access to appropriate information and practice problems at the proper time. Some general instructions are listed below.

   a. The first thing you must do as you start each subsection is to elect whether or not to take the pretest. If you take the pretest and make the required passing grade, you will be allowed to bypass the instructional and practice frames (exercises) designed to help you learn that subject matter.

   b. If you elect to bypass the pretest or do not attain a passing grade on it, you will be given an explanation of a basic concept, then required to apply this concept in the solution to some problems.

   c. After you have completed all of the instructional and practice frames for a subsection you will be given a test on that subject and then directed to continue with the next subsection.

   d. As you progress through this programmed text, read the explanations, complete the problems and then check your answers. If you answer incorrectly to any practice problem, turn back and reread the explanation to determine the reason for your error.

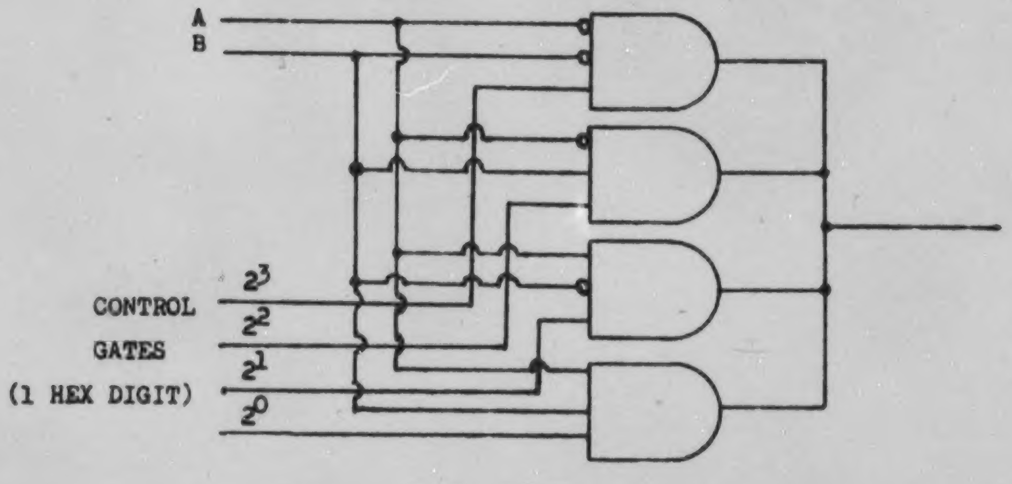
   e. You can obtain additional practice for each section by turning back to the pretest and completing it. Specific instructions for this procedure will be given at appropriate points.

TURN TO PRETEST I ON PAGE 1 AND BEGIN

TRUTH TABLE

	Operand bit combinations A B 0 0	 A B 0 1	 A B 1 0	 A B 1 1	Symbolic representation of logical function
Result	0	0	0	1	A·B
Bit	0	1	0	0	A'·B
For	0	0	1	0	A·B'
Each	0	1	1	1	A+B
Combination	1	1	0	1	A'+B
	1	0	1	1	A+B'
	0	1	1	0	A(Exclusive OR)B
	1	0	0	0	NOR
	1	1	1	0	NAND
	1	0	0	0	A'·B'
	1	1	1	0	A'+B'

LOGIC DIAGRAM



PRETEST I

310

INSTRUCTIONS: If you understand and believe you can solve problems using the AND logic function, continue below. If not, turn to frame #1 on page 3.

1. Place a checkmark beside each of the following symbols that indicate the AND functions.

a. _____ A+B=C

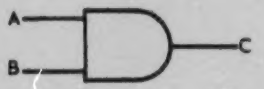
b. _____ A ∧ B=C

c. _____ AB=C

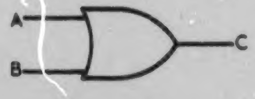
d. _____ A ∨ B=C

e. _____ A·B=C

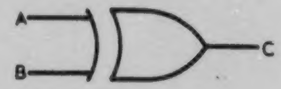
f. _____



g. _____



h. _____



2. Perform the AND function for each of the following problems. You may use the Truth Table on page iv.

a. 110
101

b. 111
101

c. 11111
10000

d. 10101
10011

3. Place a checkmark beside the correct definition of the AND function.

a. _____ All inputs must be zeros to have a zero output.

b. _____ All inputs must be ones to have a one output.

c. _____ A one at any or all inputs will produce a one output.

d. _____ A one input with all other inputs zeros will produce a one output.

ANSWERS TO PRETEST I

1. a. ____
   b. ✓
   c. ✓
   d. ____
   e. ✓
   f. ✓
   g. ____
   h. ____

2. a. 100 b. 101 c. 10000 d. 10001

3. b. ✓

INSTRUCTIONS: The maximum error allowed for satisfactory completion of PRETEST I is one incorrect answer in part 1, 2, or 3, or two incorrect answers on the entire test. If you met this requirement, turn to page 11 and continue; otherwise turn to page 3 and continue.

Frame 1

The logic used by a computer in solving a problem consists of a set of rules. In order to write programs to cause the computer to solve specific problems, the programmer must be able to use the computer logic rules to solve sample problems.

There are three basic logic circuits in a computer; therefore, three basic logic functions the computer can perform. The three logic functions are called AND, OR, and NOT.

Each of these logic functions has one or more unique symbols used in an equation to denote that specific function.

The symbols used in a logic equation to denote the AND function are the same as those used to denote multiplication in an arithmetic equation with one additional symbol used by some writers. This symbol is an inverted V ($\wedge$).

Place a checkmark beside each of the following equations that denote the AND function.

a. ____ $X+Y=Z$

b. ____ $R \cdot M=N$

c. ____ $AB=C$

d. ____ $A+B+C=D$

e. ____ $XYZ=A$

f. ____ $R \wedge S=T$

g. ____ $D \vee E=F$

h. ____ $X+Y+Z=A$

Check your answers against those at the top of the following page.

Answers to Frame 1

a. ____

b. ✓

c. ✓

d. ____

e. ✓

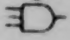
f. ✓

g. ____

h. ____

Frame 2

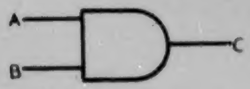
A writer will normally use only one symbol to denote a particular function. However, you will be reading technical literature written by many different writers; therefore, you should be able to recognize the symbol in all of its usual forms.

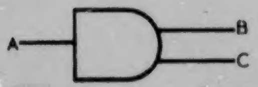
In your job as a programmer, you may also be exposed to some logic diagrams. The symbol for the AND function in a logic diagram appears like an elongated D ( ) with two or more inputs and one output ().

Place a checkmark beside each of the following that correctly denotes the AND function.

a. ____ A+B=C

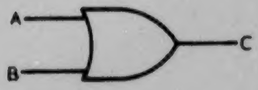
f. ____ A∨B=C

b. ____ 

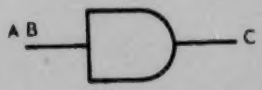
g. ____ 

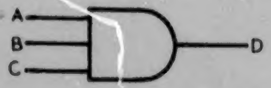
c. ____ A·B=C

h. ____ A∧B=C

d. ____ 

i. ____ AB=C

e. ____ 

j. ____ 

Check your answers at the top of the following page.

Answers to Frame 2

a. ____

b. ✓

c. ✓

d. ____

e. ____

f. ____

g. ____

h. ✓

i. ✓

j. ✓

Frame 3

The purpose of a logic circuit in the computer is to combine values within the computer according to the rules of the logic function being performed. The values to be combined are stored in registers and are combined one digit at a time. Since the computer uses binary numbers, the only values it has are ones and zeros.

The rule for the AND function is "an output is produced when and only when all inputs are present." This rule can be converted to a Truth Table as follows:

A X B = C
0 X 0 = 0
0 X 1 = 0
1 X 0 = 0
1 X 1 = 1

This Truth Table demonstrates the rule or, stated in reverse, if a zero is present at any or all inputs there is a zero or no output.

Use the Truth Table above to help you solve the following number combinations using AND logic. Remember to combine the digits in corresponding positions one at a time.

a.	1 0	b.	0 0	c.	1 1	d.	0 1
e.	11 10	f.	101 100	g.	001 110	h.	10101 11001

Check your answers.

Answers to Frame 3

a. 0 b. 0 c. 1 d. 0

e. 10 f. 100 g. 000 h. 10001

Frame 4

a. Place a checkmark beside the correct definition of the AND function.

(1) _____ A one at any or all inputs will produce a one output.

(2) _____ A one input with all other inputs of zero will produce a one output.

(3) _____ All inputs must be zeros to have a zero output.

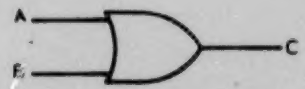
(4) _____ All inputs must be ones to have a one output.

b. Solve the following problems using the AND function.

(1)	101	(2)	101	(3)	10000	(4)	10011
	110		111		11111		10101

c. Place a checkmark beside each of the following symbols that indicates the AND function.

(1) _____



(5) _____ A+B=C

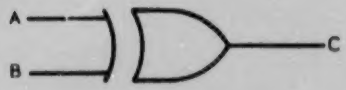
(2) _____ A∨B=C

(6) _____ A∧B=C

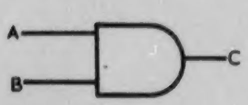
(3) _____ AB=C

(7) _____ A·B=C

(4) _____



(8) _____



Answers to Frame 4

a. (4) ✓

b. (1) 100 (2) 101 (3) 10000 (4) 10001

c. (1) ____ (5) ____

(2) ____ (6) ✓

(3) ✓ (7) ✓

(4) ____ (8) ✓

INSTRUCTIONS: If you need or want additional practice on this section, turn back to page 1 and solve the problems in PRETEST I. If not, turn to page 11 and continue.

PRETEST II

INSTRUCTIONS: If you understand and believe you can solve problems using the OR logic function, continue below. If not, turn to Frame 5 on page 13.

1. Place a checkmark beside each of the following symbols that indicates the OR function.

a. _____ $A+B=C$

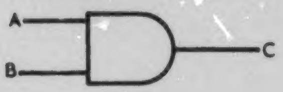
b. _____ $A \wedge B=C$

c. _____ $AB=C$

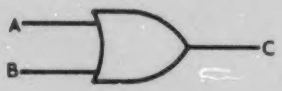
d. _____ $A \vee B=C$

e. _____ $A \cdot B=C$

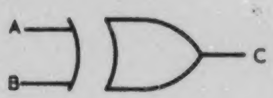
f. _____



g. _____



h. _____



2. Perform the OR function for each of the following problems. You may use the Truth Table on page iv.

a. 110
101

b. 111
101

c. 11111
10000

d. 10101
11001

3. Place a checkmark beside the correct definition of the OR function.

a. _____ All inputs must be ones to have a one output.

b. _____ All inputs must be zeros to have a one output.

c. _____ A one output will be produced only when there is a one at one input and all other inputs are zeros.

d. _____ A one at any or all inputs will produce a one output.

ANSWERS TO PRETEST II

1. a. ✓ e. ____
b. ____ f. ____
c. ____ g. ✓
d. ✓ h. ____

2. a. 111 b. 111 c. 11111 d. 11101

3. d. ✓

INSTRUCTIONS: The maximum error allowed for satisfactory completion of PRETEST II is one incorrect answer in part 1, 2, or 3, or two incorrect answers on the entire test. If you met this requirement, turn to page 21 and continue; otherwise turn to page 13 and continue.

Frame 5

The next logic function we will study is the OR function. The OR function is indicated in logical equations by the same symbols used for addition in mathematical equations. However, some writers use a symbol that looks similar to a V.

Write OR beside each of the following equations that indicates the OR function and AND to identify equations that indicate the AND function.

a. _____ $X+Y+Z=A$

b. _____ $D \vee E=F$

c. _____ $R \wedge S=T$

d. _____ $XYZ=A$

e. _____ $A+B+C=D$

f. _____ $AB=C$

g. _____ $R \cdot M=N$

h. _____ $X+Y=Z$

Answers to Frame 5

a. OR

b. OR

c. AND

d. AND

e. OR

f. AND

g. AND

h. OR

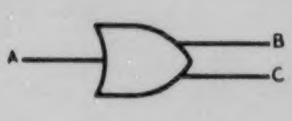
Frame 6

The logic circuit symbol for the OR function is similar to the AND logic circuit symbol except that the input end of the symbol is concave. The OR logic may have two or more inputs and one output.

Place a checkmark beside each of the following that correctly denotes the OR logic function.

a. _____ $A \vee B = C$

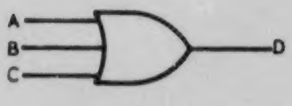
b. _____



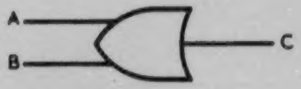
c. _____ $A \wedge B = C$

d. _____ $AB = C$

e. _____

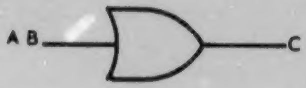


f. _____



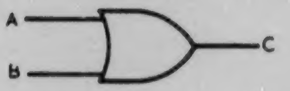
g. _____ $A + B = C$

h. _____



i. _____ $A \cdot B = C$

j. _____



Answers to Frame 6

a. ✓
b. ____
c. ____
d. ____
e. ✓
f. ____
g. ✓
h. ____
i. ____
j. ✓

Frame 7

When we combined values using AND logic, we found that a zero at any or all inputs would produce zero output. When using OR logic to combine values, a one at any or all inputs will produce a one output.

This rule can be converted to a Truth Table as follows:

A + B = C
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 1

Use the Truth Table above to help you solve the following number combinations using OR logic. Remember to combine the digits in corresponding positions one at a time.

a.	1 0	b.	0 0	c.	1 1	d.	0 1
e.	11 10	f.	101 100	g.	001 110	h.	10101 11001

327

Answers to Frame 7

a.	1	b.	0	c.	1	d.	1
e.	11	f.	101	g.	111	h.	11101

320

Frame 8

a. Place a checkmark beside the correct definition of the OR function.

(1) _____ A one output will be produced only when there is a one at one input and all other inputs are zeros.

(2) _____ A one at any or all inputs will produce a one output.

(3) _____ All inputs must be ones to have a one output.

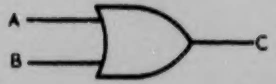
(4) _____ All inputs must be zeros to have a one output.

b. Solve the following problems using the OR function.

(1)	101	(2)	101	(3)	10000	(4)	10011
	110		111		11111		10101

c. Place a checkmark beside each of the following symbols that indicates the OR function.

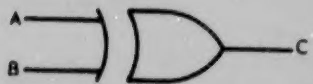
(1) _____



(2) _____ $A \vee B = C$

(3) _____ $AB = C$

(4) _____

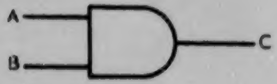


(5) _____ $A + B = C$

(6) _____ $A \wedge B = C$

(7) _____ $A \cdot B = C$

(8) _____



Answers to Frame 8

a. (2) ✓

b. (1) 111 (2) 111 (3) 11111 (4) 10111

c. (1) ✓ (5) ✓

(2) ✓ (6) ___

(3) ___ (7) ___

(4) ___ (8) ___

INSTRUCTIONS: If you need or want additional practice on this section, turn back to page 11 and solve the problems in PRETEST II. If not, turn to page 21 and continue.

PRETEST III

INSTRUCTIONS: If you understand and believe you can solve problems using the NOT logic function, continue below. If not, turn to Frame 9 on page 23.

1. Place a checkmark beside each of the following symbols that contains a NOT function.

a. ____ A+B=C

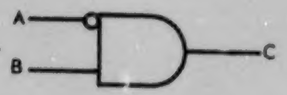
b. ____ A∧B'=C

c. ____ AB=$\overline{C}$

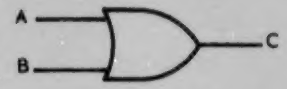
d. ____ $\overline{A}$∨B=C

e. ____ A·B=C

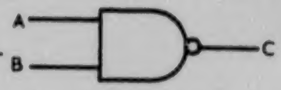
f. ____



g. ____



h. ____



2. Perform the logic functions indicated for each of the following problems. You may use the Truth Table on page iv.

a. A+$\overline{B}$=C - A=10101
B=10001
C=

b. A·B=C' - A=11001
B=10101
C=

c. A'∨B=C - A=11100
B=10110
C=

d. A∧$\overline{B}$=C - A=10011
B=10101
C=

ANSWERS TO PRETEST III

1. a. ____
   b. ✓
   c. ✓
   d. ✓
   e. ____
   f. ✓
   g. ____
   h. ✓

2. a. 11111
   b. 01110
   c. 10111
   d. 00010

INSTRUCTIONS: The maximum error allowed for satisfactory completion of PRETEST III is one error in part 1 and one error in part 2. If you met this requirement, turn to page 37 and continue; otherwise, turn to page 23 and continue.

Frame 9

The third basic logic function performed by a computer is NOT logic. This is often called an inverter because it inverts the signal; i.e., if the input is 1, the output is 0.

NOT logic is used in conjunction with AND and OR logic in most computer circuits. The NOT function can be placed in either the input or the output to the AND or OR logic device and is performed in addition to the AND or OR function; i.e., if the output from an AND function based on the inputs would normally be a 1, an inversion (NOTing) of this output would make it a zero.

The NOT function in an equation is shown by a line above the expression or a prime beside it.

Place a checkmark beside each of the following equations that contains a NOT function.

a. _____ $\overline{A}+B=C$

b. _____ $A \cdot B=C$

c. _____ $A \vee B'=C$

d. _____ $A \wedge B=C'$

e. _____ $AB=C$

f. _____ $ABC=\overline{D}$

g. _____ $A+B=C$

h. _____ $A+B+C=D$

Answers to Frame 9

a. ✓

b. ____

c. ✓

d. ✓

e. ____

f. ✓

g. ____

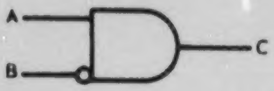
h. ____

Frame 10

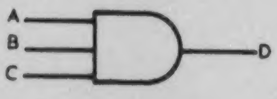
The NOT function is indicated in logic circuits by a small circle interrupting the line used to show an input or output of an AND or an OR logic symbol.

Place a checkmark beside each of the following diagrams that contains a NOT symbol.

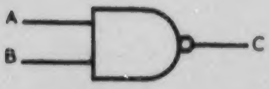
a. ______



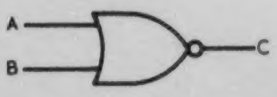
e. ______



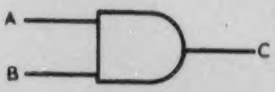
b. ______



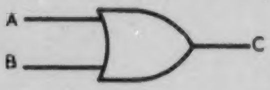
f. ______



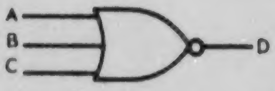
c. ______



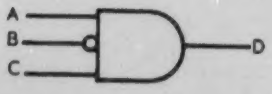
g. ______



d. ______



h. ______



Answers to Frame 10

a. ✓

b. ✓

c. ____

d. ✓

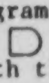
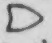
e. ____

f. ✓

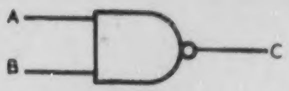
g. ____

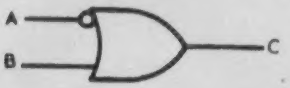
h. ✓

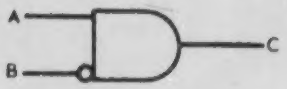
Frame 11

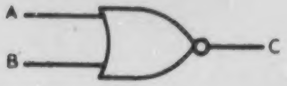
The logic diagrams can be matched to the equations by matching the basic symbols (·= ) (+= ), and then matching the circle at the input or output with the line above or prime beside one of the factors or the result shown in the equation.

Match the logic diagrams to the equations by writing the letter that identifies each diagram into the space beside the appropriate equation.

a. 

b. 

c. 

d. 

(1) _____ $A\cdot B'=C$

(2) _____ $AB=\overline{C}$

(3) _____ $A+B=C'$

(4) _____ $\overline{A}+B=C$

337

Answers to Frame 11

(1) c

(2) a

(3) d

(4) b

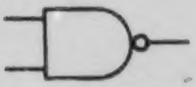
330

Frame 12

When we combine logic functions AND and NOT in that order we produce what is called a NAND (NOT-AND) function. We can produce a NOR function using the same principle.

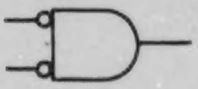
Do not be confused by inverted inputs because the results are not the same; i.e., using the values 110 and 101, observe the results below:

$A \cdot B = C'$
or



110
101
100 Invert 011

$A' \cdot B' = C$
or

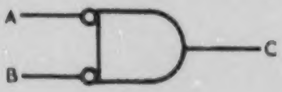


110 Invert 001
101 Invert 010
000

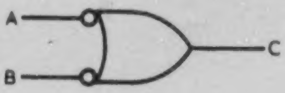
A simple procedure for solving problems calling for NAND logic or NOR logic is - solve for the basic function, AND or OR; then invert the result. Note the left-hand example above. If you use the truth table it gives you the correct result without inversion.

Write AND, OR, NAND, and NOR beside each of the following, as appropriate, for identification.

a. ____



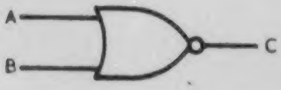
e. ____



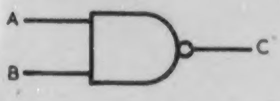
b. ____ $A' \cdot B' = C$

f. ____ $A+B=\overline{C}$

c. ____



g. ____



d. ____ $A+\bar{\bar{B}}=C$

h. ____ $A \wedge B = C'$

Answers to Frame 12

a. AND

b. AND

c. NOR

d. OR

e. OR

f. NOR

g. NAND

h. NAND

Frame 13

The Truth Table shown below is a combination of AND, OR, NAND, and NOR. Inputs (values of A and B) are shown at the top of each column and the output (value of C) in the box at the intersection of the appropriate line and column. The equation is shown at the left of each line.

Label the lines AND, OR, NAND, and NOR to identify the line showing the Truth Table for each of these logic functions.

	0-0	0-1	1-0	1-1
A·B=C'	1	1	1	0
A+B=C'	1	0	0	0
A+B=C	0	1	1	1
A·B=C	0	0	0	1

Logic Function

a. ______________

b. ______________

c. ______________

d. ______________

Answers to Frame 13

a. NAND

b. NOR

c. OR

d. AND

Frame 14

Now let's try using the combination Truth Table to solve problems. Solve each problem using AND, OR, NAND, and NOR logic functions.

	0-0	0-1	1-0	1-1
A·B=C	0	0	0	1
A+B=C	0	1	1	1
A+B=C'	1	0	0	0
A·B=C'	1	1	1	0

a. 10101 / 11001 AND

10101 / 11001 OR

10101 / 11001 NAND

10101 / 11001 NOR

b. 11111 / 10101 AND

11111 / 10101 OR

11111 / 10101 NAND

11111 / 10101 NOR

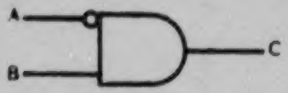
Answers to Frame 14

a.	AND 10001	OR 11101	NAND 01110	NOR 00010
b.	AND 10101	OR 11111	NAND 01010	NOR 00000

Frame 15

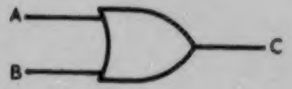
a. Place a checkmark beside each of the following symbols that contains a NOT function.

(1) ____ $A \cdot B = C$

(2) ____ 

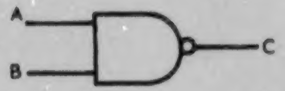
(3) ____ $\bar{A} \vee B = C$

(4) ____ $AB = \bar{C}$

(5) ____ 

(6) ____ $A \wedge B' = C$

(7) ____ $A + B = C$

(8) ____ 

b. Perform the logic function indicated by the equation for each of the following problems. Use the Truth Table on page iv.

(1) $A + B = C'$ - A=10101
B=11001
C=

(2) $A + \bar{B} = C$ - A=10001
B=10101
C=

(3) $A \wedge \bar{B} = C$ - A=10101
B=10011
C=

(4) $A' \vee B = C$ - A=10110
B=11100
C=

Answers to Frame 15

a. (1) ____ (5) ____

(2) ✓ (6) ✓

(3) ✓ (7) ____

(4) ✓ (8) ✓

b. (1) 00010 (3) 00100

(2) 11011 (4) 11101

INSTRUCTIONS: If you need or want additional practice on this section, turn back to page 21 and solve the problems in PRETEST III. If not, turn to page 37 and continue.

PRETEST IV

INSTRUCTIONS: If you understand and believe you can solve problems using the EXCLUSIVE OR logic function, continue below. If not, turn to frame 16 on page 39.

1. Place a checkmark beside each of the following symbols that indicates an EXCLUSIVE OR function.

a. _____ $\overline{AB}+\overline{AB}=C$

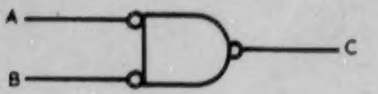
b. _____ $A \cdot B'+A' \cdot B=C$

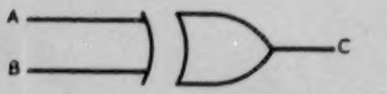
c. _____ $A \cdot B+A \cdot B=C$

d. _____ $\overline{A} \cdot B+A \cdot \overline{B}=C$

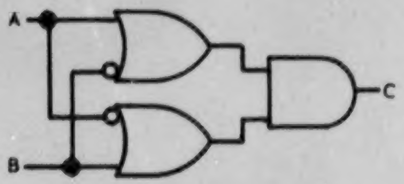
e. _____ $A'+B \cdot A+B'=C$

f. _____





g. _____



h. _____

2. Perform the EXCLUSIVE OR function for each of the following problems. You may use the Truth Table on page iv.

a. 10101
11001

b. 111
101

c. 10001
10000

d. 11111
10101

3. Place a checkmark beside the correct definition of the EXCLUSIVE OR function.

a. _____ A one output is produced only when there is a one at one input and zeros on all others.

b. _____ All inputs must be ones to have a one output.

c. _____ A one at one or more inputs will produce a one output.

d. _____ All inputs must be zeros to have a zero output.

ANSWERS TO PRETEST IV

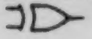
1. a. ____ f. ____
b. ✓
c. ____ g. ✓
d. ✓
e. ____ h. ____

2. a. 01100 b. 010 c. 00001 d. 01010

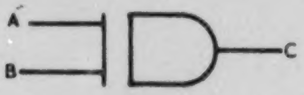
3. a. ✓

INSTRUCTIONS: The maximum error allowed for satisfactory completion of PRETEST IV is one incorrect answer in part 1, 2, or 3, or two incorrect answers on the entire test. If you met this requirement, turn to page 45 and continue; otherwise, turn to page 39 and continue.

Frame 16

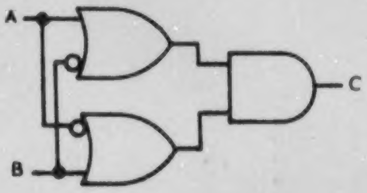
The EXCLUSIVE OR function produces a one output when one input is one and all others are zero. It uses a combination of two AND functions with alternate inputs inverted and an OR function at the output of the two AND functions. In addition to the combination of AND and OR functions symbols, the symbol  is sometimes used to indicate the EXCLUSIVE OR function.

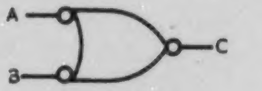
Place a checkmark beside each of the following that indicates the EXCLUSIVE OR function.

a. ____ 

b. ____ A'+B·A+B'=C

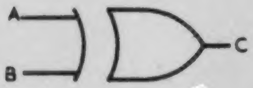
c. ____ $A\cdot\overline{B}+\overline{A}\cdot B=C$

d. ____ 

e. ____ 

f. ____ $\overline{AB}+\overline{AB}=C$

g. ____ A·B+A·B=C

h. ____ 

Answers to Frame 16

a. _____
b. _____
c. ✓
d. _____
e. _____
f. _____
g. _____
h. ✓

Frame 17

The Truth Table for the EXCLUSIVE OR function is shown in the combined Truth Table on page iv of this book. Observe that 0·0=0, 0·1=1, 1·0=1, and 1·1=0.

A simple procedure for solving problems calling for EXCLUSIVE OR logic is - observe the operands that are to be combined. If there is a 1 in any position and all other positions are 0, the result will be 1. Any other combination will produce a 0 result. If you use the truth table it gives you the correct result for each bit combination.

Use the Truth Table on page iv of this book to help you combine the following values using the EXCLUSIVE OR function.

a. 11001
10101

b. 101
111

c. 10000
10011

d. 11111
11111

e. 10101
01010

f. 10101
11111

Answers to Frame 17

a. 01100 b. 010 c. 00011

d. 00000 e. 11111 f. 01010

Frame 18

a. Place a checkmark beside the correct definition of the EXCLUSIVE OR logic function.

(1) ____ All inputs must be ones to have a one output.

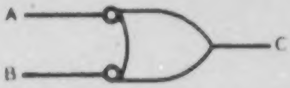
(2) ____ A one output is produced only when there is a one at one input and all other inputs are zeros.

(3) ____ All inputs must be zeros to have a zero output.

(4) ____ A one at one or more inputs will produce a one output.

b. Place a checkmark beside each of the following that indicates an EXCLUSIVE OR function.

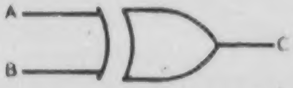
(1) ____



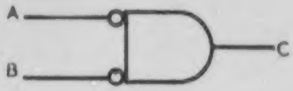
(2) ____ $A+B' \cdot A'+B=C$

(3) ____ $AB'+AB'=C$

(4) ____



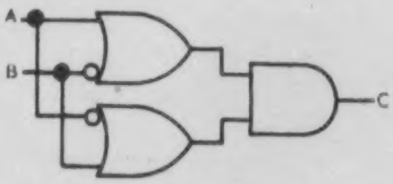
(5) ____



(6) ____ $A' \cdot B+A \cdot B'=C$

(7) ____ $A \wedge B' \vee A' \wedge B=C$

(8) ____



Answers to Frame 18

a. (2) ✓

b. (1) ____

(2) ____

(3) ____

(4) ✓

(5) ____

(6) ✓

(7) ✓

(8) ____

Frame 19

This is the final exercise in this lesson. It will review the entire lesson and allow you to check your attainment of the objective.

a. Perform the logic functions indicated for each of the following problems. Use the Truth Table on page iv.

(1) $A \cdot B = \overline{C}$ - $A=1010$
$B=\underline{1100}$
$C=$

(2) $A \cdot \overline{B} + \overline{A} \cdot B = C$ - $A=1001$
$B=\underline{1100}$
$C=$

(3) $A \wedge B = C$ - $A=11001$
$B=\underline{10001}$
$C=$

(4) $AB = C$ - $A=10111$
$B=\underline{11100}$
$C=$

(5) $\overline{A} + B = C$ - $A=1011$
$B=\underline{1110}$
$C=$

(6) $A' + B' = C$ - $A=11001$
$B=\underline{10101}$
$C=$

(7) $A' \cdot B + A \cdot B' = C$ - $A=11100$
$B=\underline{10010}$
$C=$

(8) $A \cdot B' = C$ - $A=10111$
$B=\underline{10101}$
$C=$

(9) $A' \cdot B' = C$ - $A=00111$
$B=\underline{10100}$
$C=$

(10) $A + B' = C$ - $A=1010$
$B=\underline{1100}$
$C=$

(11) $A + B = C$ - $A=11100$
$B=\underline{10110}$
$C=$

(12) $A + B = \overline{C}$ - $A=1111$
$B=\underline{1010}$
$C=$

(13) $A' \cdot B = C$ - $A=1111$
$B=\underline{1010}$
$C=$

(14) $A \vee B = C$ - $A=10101$
$B=\underline{10011}$
$C=$

b. Write the letters used to identify the appropriate logic symbol, below right, into the space beside each equation, below left.

(1) $A \cdot B = C$ ______

(2) $A' \wedge B = C$ ______

(3) $A \vee B' = C$ ______

(4) $\overline{A} \cdot \overline{B} = C$ ______

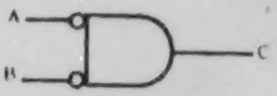
(5) $A + B = C$ ______

(6) $AB = \overline{\overline{C}}$ ______

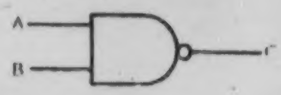
(7) $A + B = C'$ ______

(8) $A' \cdot B + A \cdot B' = C$ ______

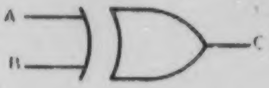
aa.



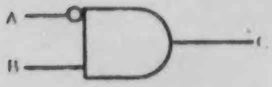
bb.



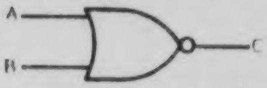
cc.



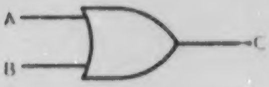
dd.



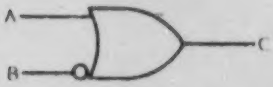
ee.



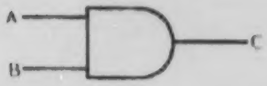
ff.



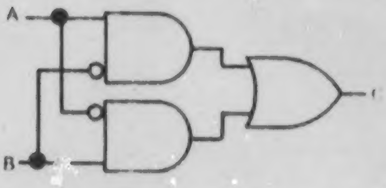
gg.



hh.



ii.



c. Write the letters used to identify the appropriate logic symbol, as illustrated on page 46, into the space beside each title below.

(1) NAND _____

(2) NOR _____

(3) EXCLUSIVE OR _____ & _____

(4) Other diagrams with NOT logic _____, _____ & _____

(5) AND _____

(6) OR _____

d. Write the letters used to identify the appropriate definition into the space beside each term below.

(1) NOT _____

(2) OR _____

(3) AND _____

(4) EXCLUSIVE OR _____

aa. An input of one on one leg will produce an output of one if all other inputs are zeros.

bb. A one input on any or all legs will produce a one output.

cc. Changes a one to a zero or a zero to a one.

dd. All inputs must be ones to produce a one output.

Answers to Frame 19

INSTRUCTIONS: Check your answers against the correct answers below. If you answered incorrectly to two or more problems in any section you should go back and review the frames that present the principles concerned. Your instructor will administer a test during class which requires 70% of the answers correct for satisfactory completion.

a. (1) 0111
(2) 0101
(3) 10001
(4) 10100
(5) 1110
(6) 01110
(7) 01110
(8) 00010
(9) 01000
(10) 1011
(11) 11110
(12) 0000
(13) 0000
(14) 10111

b. (1) hh
(2) dd
(3) gg
(4) aa
(5) ff
(6) bb
(7) ee
(8) cc & ii

c. (1) bb
(2) ee
(3) cc & ii
(4) aa, dd & gg
(5) hh
(6) ff

d. (1) cc
(2) bb
(3) dd
(4) aa

SELF-INSTRUCTION TEXT
KDA-295

Technical Training

TOP DOWN STRUCTURED PROGRAMMING

January 1978

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler [illegible]

DO NOT USE ON THE JOB

TABLE OF CONTENTS

SECTION I - TOP DOWN STRUCTURED PROGRAMMING ORGANIZATION

CHAPTER		PAGE
1	Chief Programming Team	1-1
2	Development Support Library (DSL)	2-1
3	Structured Walk-Throughs	3-1

SECTION II - TDSP DESIGN TOOLS AND TECHNIQUES/DOCUMENTATION

CHAPTER		PAGE
4	Data Flow Graphs	4-1
5	Structure Charts	5-1
6	HIPOs	6-1
7	Structured Flowcharts	7-1
8	Design Techniques	8-1

SECTION III - TOP DOWN IMPLEMENTATION

CHAPTER		PAGE
9	Top Down Design Implementation Strategy	9-1
10	Program Design Language	10-1
11	Structured Code	11-1

APPENDIX A - Glossary of TDSP Terms A-1

INTRODUCTION

During the last few years there has been a quiet revolution developing within the software profession. It has not been a revolution founded in violence or bent upon the extension of national boundaries. It has been a revolution to establish a new scientific discipline within an existing science.

But what is this revolution? Perhaps a more fundamental question would be - why a revolution? Let's discuss these questions separately.

Revolution is defined as a complete change of any kind. So to define this revolution we are talking about, we must define what is being changed. The object of change is the methodology we use to create computer programs. It focuses upon the program development process and the nature of the product resulting from our programming effort.

Top-down design and structured programming concepts are dedicated to the production of a good program. What is a good program? Since there are so many possible answers, we will list only seven desirable qualities of a program.

1. The program works - this is the most important quality of any program. It is truly incredible to see the number of times that two programmers with similar programs will debate the following: "My program is nine times faster than yours - and it requires four times less memory!". . . to which the other programmer replies, "Yes, but your program doesn't work, and mine does!" In the book The Element of Programming Style by Kernighan and Plauger, three rules are cited as follows:

- Make it right before you make it faster.
- Make it failsafe before you make it faster.
- Make it clear before you make it faster.

Harlan D. Mills substantiates these rules in his book, How to Write Correct Programs and Know It:

"There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire the confidence that a program has no errors is never to find the first one, no matter how much it is tested and used. It is an old myth that programming must be an error-prone, cut-and-try process of frustration and anxiety. But there is a new reality that you can learn to consistently write programs which are error-free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess."

2. Lower testing costs - Thirty to fifty percent of the total project time is devoted to testing and debugging computer programs.

3. Lower maintenance costs - Surveys show that the average organization spends at least 50 percent of its entire EDP budget on the maintenance of existing systems. In one Air Force Command, seventy percent of its software costs is required for maintenance. The high cost of maintenance limits the development of new software.

4. Ease of modification - Regardless of the amount of preparation and planning, there will always be required changes and modifications for programs. Every segment or module of a program should be designed with an eye toward its eventual revision or modification.

5. Lower development costs - In TDSP, the modules of a program are tested and implemented as they are developed. This greatly reduces the cost of development since the time normally required for implementation is eliminated.

6. Uncomplicated design - Programs need not be complex to work properly. In fact, the most logical way, and in some cases, the only way to make a program easy to test, maintain, and modify is to keep it simple. In other words, always design a program so that someone else can maintain it. One survey revealed that a program is usually maintained by ten different programmers.

7. Efficiency - Efficiency considerations are undertaken when they are not even needed. A recent study revealed that in a typical FORTRAN program, fifty percent of the execution time was used by only three percent of the instructions. These findings indicate that the following programming methodology should be adopted:

a. Write the module/program in a straightforward manner, emphasizing simplicity and reliability.

b. After the module/program is working, rewrite and optimize the time-consuming code.

Using this method, the programmer will invest his time in optimizing only those instructions which will save an appreciable amount of execution time.

Now let's discuss the second question:

What caused this revolution? The answer is money. Today, the cost of software is about three times the cost of hardware. The Air Force is spending approximately five percent of its total annual budget for software. If software costs continue to increase, the Air Force will be spending at least ninety percent of its EDP dollars for software in 1985.

Maintenance costs, including modifications, normally range as high as 50 to 100 percent of the initial system development cost. To decrease the cost of software, we must reduce the costs of initial development and continuing maintenance and increase the reliability of a system. TDSP was developed to reduce software costs while increasing software reliability.

Top-Down Structured Programming (TDSP) techniques constitute a methodology which provide a systematic approach to problem solving in software development. This technology is applicable to many current and future Air Force programs. The TDSP methodology utilizes a combination of tools and techniques which are defined by the Air Force to include:

- Top-Down Design
- Top-Down Documentation
- Top-Down Implementation
- Structured Coding
- Development Support Library
- Structured Walk-throughs
- Program Design Language
- Chief Programming Team

These techniques/tools are defined as follows:

1. Top-Down Design. Top-down design is the process of decomposing a single complex function into hierarchical levels of simpler functions. The levels of the hierarchy correspond to the control levels of the tasks performed by the system components. Each level of the software design shall be logically complete in itself. The top level contains the highest level of control logic and decisions within the software design. Each sublevel is a self-contained component whose operation is subordinate to the next higher level.

2. Top-Down Documentation. Top-down documentation illustrates the top-down design and is delivered in increments as the system is developed. It records top-down design decisions, insures that those decisions are made prior to coding, and serves as the review/approval document that authorizes coding. It includes descriptions, specifications, graphical representations, manuals, plans, reports, listings, and other technical documents. Some techniques of documentation currently available are: hierarchical plus input-process-output (HIPO) charts, structured design charts, program design languages, and structured flow charts.

3. Top-Down Implementation. Top-down implementation is the coding, verification, and implementation of higher levels of the system logic prior to the coding of any subordinate modules. Lower level modules, needed for an interface, are coded as dummy code (program stubs) which need not perform any meaningful computations. For example, the program stub may output a message for debugging purposes or perhaps simulate the expected output each time it is executed. When the upper level coding has been verified, actual coding is substituted for program stubs at successively lower levels of logic until the entire system has been implemented and verified. The important point is that program modules at each level are fully integrated and verified with their predecessors before coding begins on the next lower level.

4. Structured Coding. Structured coding is the writing of programs by repeated use of predefined control logic primitives: Sequence, IFTHENELSE, DOWHILE, DOUNTIL, and CASE. Each of these will be explained later in this publication.

5. Development Support Library. The development support library serves as a central repository of all data relevant to the project, in both human-readable and machine-recognizable form. As such, it is used to aid in the organization and control of the developing software. It is the focal point of information exchange - both management and technical - for the life of the project.

6. Structured Walk-throughs. Structured walk-throughs are technical examinations of the design, implementation, and documentation to provide positive feedback to the programmer. These walk-throughs are scheduled by the programmer and attended by his peers.

7. Program Design Language. Program design language is a language for describing the control structure and general organization of a computer program. It is an English-like representation of a procedure which is easy to read and comprehend. It is structured in the sense that it utilizes the predefined control logic primitives. Indentation is used to make the PDL easier to read. This technique facilitates the translation of functional specifications into computer instructions using top-down design and structured coding.

8. Chief Programming Team. The chief programming team is minimally defined as "two or more programmer/analysts assigned to a project who possess a combined responsibility for the quality of the delivered product."

This introduction has summarized the techniques of TDSP and outlined the target qualities of a good program. In the following chapters, do not look for absolute, concrete statements on the right or wrong way to utilize a technique. However, integrate these concepts with your own, utilizing individual expertise, to give the user a program that works! works everytime! does what it is supposed to do! is easily modified and can be maintained easily.

SECTION 1

TOP-DOWN STRUCTURED PROGRAMMING ORGANIZATION

CHAPTER 1

CHIEF PROGRAMMING TEAM

While studying this chapter you should learn:

1. Purpose of Chief Programming Team (CPT).
2. Three main members of the CPT.
3. Duties of the chief programmer.
4. Purpose of the backup programmer.
5. Function of the programming librarian.
6. Duties of short term support members.
7. Function of the project administrator.
8. Qualifications required for members of the CPT.

CHIEF PROGRAMMER TEAMS

The Chief Programmer Team (CPT) introduced in 1969 is a different technique for managing the software development process. Its basic idea is to organize a small number of highly competent people according to their special and complementary skills. The newly developing structured programming technology provides the vehicle by which team members communicate and function. With this combination of people and technology, comprehension of a fairly complex design and implementation can reside in one person. Significant advantages are reported as a result of the simplifications and unified structure which can be achieved by one mind in designing an entire system. These advantages are manifest primarily in improved system reliability, and the team environment has also shown an increase in productivity as measured by the number of lines of code produced per programmer day.

Improvements have been noted using the team in development efforts where the delivered product was less than 100,000 lines of high level code. On such jobs one team, consisting of from three to eight people, was adequate. Team member experience level has been varied to match job complexity with considerable success. An extension of the team concept is the hierarchy of teams. Its purpose is to organize the work force needed to produce a large, complex software system (over 100,000 lines of code) and still obtain the improvements characteristic of the team technique. Testing of the hierarchy of the teams is in progress, but the results will not be known for several years.

The CPT is primarily a management organization. As such, it exercises control over people by defining their functions and the ways in which they interrelate. It also prescribes the minimal set of tools necessary to perform the team's work. The principles and procedures of the CPT were originally described by Harlan D. Mills in June 1971. One of the first practical applications of this organization was undertaken and later described by F. T. Baker in his article, "Chief Programmer Team Management of Production Programming."

The CPT has, at its core, three members: the chief programmer, the backup programmer, and the programming librarian. These three persons perform different facets of the one job system development. They do not act independently but rather in concert on jobs that support and complement each other. The chief programmer's role is the result of recognizing that complex system and in-depth programming knowledge are a necessary combination. The position of the backup programmer recognizes the need for idea refinement through peer review, and also continuity in supervision and decision making in the absence of the chief programmer. The programming librarian role is the result of separating to a great degree the clerical and technical activities in the programming process.

The chief programmer is a manager and all other members of the team report directly to him. However, his principal job is to design, code, and test programming systems. If the chief programmer does not manage the team, he will lose control of the production of the system. If he does not actively program and provide technical leadership, he will certainly lose control of the programming activity itself. The chief programmer functions as the technical, but not necessarily the administrative, manager for the team members. As technical leader, he is personally responsible for the complete design of a substantial software system. He also writes the critical code of the system and directly supervises implementation of the remaining portions by team members. The chief programmer reads and constructively criticizes all of the program code developed by the team. The chief programmer always directs an organization at the first line of management. He is concerned primarily with, and contributes to, the production of the software system.

The chief programmer is responsible for the direction and supervision of team members. The chief programmer identifies and apportions assignments, constructively criticizes progress on design and coding. Because of the chief programmer's close working relationship with team members, he is well qualified to fulfill these duties. The management task of the chief programmer is a great deal simpler than managing a contemporary project.

Structured programming standards permit the chief programmer to read, understand, validate, evaluate, and appraise all program data developed by the team. This visibility of the manager motivates better programming throughout the team.

The chief programmer position is designed to center performance responsibilities on the most highly qualified persons. Responsibility is clearly identified; authority, through the management title, is also clearly identified and permits timely decision making by the most well informed person. The individual duties of the chief programmer are highly structured, but the broad scope of activity provides flexibility in carrying out the work of software development. Structure increases higher level management control by increasing the visibility of the work of the responsible person. The flexibility involved offers a challenge for the exceptionally competent technical person worthy of his abilities.

The backup programmer functions in almost as critical a role as the chief programmer. He is an alter ego to the chief programmer. The backup programmer becomes totally familiar with the developing project and its rationale. He is a sounding board for the chief programmer and also contributes to design solutions and implementation techniques. He may provide independent test planning and perform as a research assistant for the chief programmer in areas of programming strategy and tactics. He most probably will contribute significant portions of the programming effort and, along with the chief programmer, reads and critiques the code of other team members.

The backup programmer also limits the management exposure brought about by the concentration of technical systems knowledge in the chief programmer. He can substitute for the chief programmer in an emergency. Because of the close working relationship and code review practices, both understand all the code produced by the team.

Technically, the backup programmer is a peer of the chief programmer. He is capable of assuming project responsibility and maintaining continuity of the development effort should the chief programmer become unavailable. Administratively, the backup programmer bears no responsibility other than those planning and review activities which the chief programmer designates. In the event the backup programmer must take over the chief programmer role, he must be ready to assume the full administrative load of the position. The transition is eased considerably by his regular participation in the decision-making processes. As with the chief programmer role, the backup programmer position is highly structured but flexible. The result is a formidable, motivating challenge to a competent technical performer.

The programming librarian is an integral member of the team. Harlan Mills points out, "The main function of a programming librarian is not to save programmers clerical work, but to maintain the status of program and test data in such a form that programmers can work more effectively..." However, the advantages of cost and accuracy of having the clerical tasks performed by clerically trained individuals instead of programmers is obvious. Programmers construct the software system by coding new programs and data on coding sheets or by altering the listings of programs in some state of completion. These external, hard copy records are transferred to internal machine readable forms by the librarian following a set of interlocking office and machine procedures. The librarian is also depended upon for all assembly, compilation, linkage editing, and test runs required on the project. The results of these runs are filed by the librarian in notebooks and archival journals which represent the current status and previous history of the project.

A software development project's records also include the volumes of system documentation. The librarian's workload is well balanced over the life of the project and requires the full range of secretarial skills plus those additional ones for maintaining machine readable files. Preserving the project's records according to specific procedures is a management defined responsibility. Therefore, the librarian, while reporting to a chief programmer, has overriding professional accountabilities as well, just as a comptroller has financial recordkeeping accountabilities that override any specific reporting relationship.

On a complex system development job, the chief programmer team core members may well require additional support for the programming and project management activities. Such requirements are recognized and planned for by the chief programmer. People who serve on the team in a support role are chosen for their special skills and perform a specific job as defined by the chief programmer. Their period of service may range from a few months to something just under the length of the project. Not all support members of the team will have the chief programmer as a manager. Those who serve for short periods of time or who act primarily as consultants report to the chief programmer technically, but not necessarily administratively. Nevertheless, communication and discipline are fostered through clear definition of responsibilities and strict adherence to the use of project standards and tools.

Typical among the short term, part-time support members of a team are finance and contract specialists. A programmer with detailed knowledge in a specific area, for example I/O terminal communications, might also be a short term support specialist. Long term members would include a project manager or administrative assistant to the chief programmer, analyst, and two or three programmers.

The programmers are expected to be technically competent and capable of producing one or more of the subsystems or major components identified as a result of the chief programmer's work. Their major function is one of implementation. They are obliged to use the structured programming technology in their work and to incorporate their work product into the developing system under the supervision of the chief programmer. Continuing dialog with the chief programmer and other team members as appropriate speeds the timely transfer of information. Despite the close working conditions and interdependencies, wide latitude in program design and development are afforded the programmers within the team effort. By a delicate balance of informed awareness, delegation of tasks and attention to discipline, the chief programmer through the team environment can provide a challenging and professional work climate for these team members.

A competent analyst offers special skills to the chief programmer in the first phase of development. In some instances analytical services may be required over the life of a complex project. It can be expected that requirements will change and possibly new ones added as the user and developer better understand the nature of the task. The analyst functions to prepare the groundwork for the system design. At the beginning of the project he works with the chief programmer. After development starts, the analyst will most probably evaluate requirement changes and their system implications for the chief programmer, keeping him advised and assisting with change planning. At the point where the system design is not susceptible to further change, the analyst's job may turn to preparing user manuals and planning for system turnover training and initial operations.

Project administration requires more time than the chief programmer has to give on a large development effort. A project administrator who works with the chief programmer can best supply these needs. The administrator may be a project manager to whom the chief programmer is responsible or he may be an administrative assistant to the chief programmer. The amount of work will vary with the size of the project and the critical nature of the scheduling with the administration full or part time to meet the work requirements.

The administrator's prime function is to monitor the nontechnical transactions of the project. It is his responsibility to prepare and oversee the budget, track labor and other charges, and audit project performance against planned schedule. He also produces the necessary reports both written and verbal for his management as well as the project's customer. Additionally, an administrator will perform some of the personnel work for the team. For example, when the team travels, the administrator will process expense accounts and coordinate relocations. The administrator must work closely with the chief programmer keeping him informed of the project's financial status and provide much of the administrative management of the project.

The CPT operation presents new challenges to the traditional management structure, as team requirements alter external as well as internal relationships. To a great extent, a team's proper functioning depends upon an agreeable accommodation by its parent organization to relax certain managerial constraints.

Instead of being a well-integrated member of a pyramidal organization, the CPT operates in a more independent fashion. Decision making, formerly decentralized over several people, is now concentrated in the chief programmer who is technical leader and project manager. While greater control within the team implies less from without, it does not mean no external control at all. The parent organization regulates the use of resources and the chief programmer is accountable to it for performance.

The CPT has some different implications for personnel management than the traditional organization. The team has a slow buildup of people at the beginning of a project and a relatively low requirement for numbers of people over its life. Because of its small size, it must have reliable sources from which to obtain people with the right qualifications at the right time. It would be appropriate perhaps for the team to work within a much larger and more functionally organized structure, one which could provide for the overlapping of work assignments. Larger organizations are also suited to providing a sufficient diversity of work assignments so that new programmers to the field are able to gain experience and expertise.

The title of chief programmer is a transient one. To qualify for the title one not only has to have the right skills but also an actual project. When the project is finished, the position of chief programmer disappears with it. There may not be new work which requires a CPT. The problems of reassigning a former chief programmer to other work are not yet fully understood and certainly not yet resolved. Much more experience with these teams will have to provide answers to these questions and also to those of organization and personnel management.

The CPT organization, then, challenges traditional hierarchical software development organizations but at the same time offers the potential of significant advantages.

The core members of the chief programmer team are exceptional people. The depth and breadth of the technical background required of the core programmers are seldom found outside the senior ranks of the profession. Even among this group few retain their technical skills and desire to mix technical and administrative work. Some observations on qualifications of core members are presented below.

The chief programmer team relationships are continuously being tested throughout the Department of Defense with satisfactory results. These prestructured CPT relationships allow the chief programmer and other team members to look outward to user needs and technical possibilities, rather than inward. This freedom to concentrate on a user's requirements is a major asset of a chief programmer team.

The chief programmer's responsibilities are successfully discharged by a person whose qualifications cover a broad spectrum of technical and managerial competence. Qualifications can be described in terms of previous experiences and demonstrations of ability. Some can be translated into job descriptions with relative ease. Other qualifications are personal attributes that are evaluated on a subjective basis, such as creative ability. These personal qualities are every bit as important to a chief programmer's success as his technical expertise, and while they do not subscribe easily to measurement, they can be recognized in the excellence of past performance.

Since the chief programmer is the technical leader of the team, he must have practical experience as a programmer and have acquired a substantial knowledge in several programming languages, operating systems, and hardware lines. To be active in system and program design as well as implementation techniques implies the need to be current in the latest developments in computer science and programming technology. Currency may be achieved through formal education, self-study, or a combination of the two. Most successful chief programmers rely on the latter. The team's dependency on top-down structured programming and the development support library makes knowledge of these techniques and skill in their application a necessity.

A chief programmer must be a participant in the managerial structure with sufficient experience to understand the software development cycle and its requirements. Proposal and project management assignments with a variety of systems and customers create the environment to acquire the proper background for learning the activities involved in software development and for learning and improving management skills.

A number of personal characteristics set chief programmers apart from their peers. Successful chief programmers are significantly more capable in judgment and decision making both technically and managerially. There is also evidence that the complexities of technical problems stimulate the chief programmer to solutions that reflect analytic and creative abilities far above average. The number of personal interactions both with the team and outside the team requires the chief programmer be especially sensitive to the intent and understandings of others as well as articulate in conveying his position to those with whom he deals. The chief programmer combines these qualities with a notable ambition to exercise the leadership role within a team. This last quality is significant, for without it the chief programmer on a complex system project stands little chance of success.

The backup programmer must share essentially the same technical and managerial experience and personal characteristics as the chief programmer since the backup programmer must be able to replace the chief programmer if necessary. If there is a distinction in knowledge and experience between the two, it may be only that the backup programmer has not the breadth found in the chief programmer. Although the backup programmer as a member of a CPT does not manage, he should have had first-line managerial experience and typically will have seen assignments in two or three technical areas as a group leader. On the personal side, he must be able to work closely, intently, and compatibly with the chief programmer while at the same time cultivating those leadership qualities that will enable him to perform eventually as a chief programmer. Thus, the chief programmer must play a dominant role in the selection of his backup.

The programming librarian needs a set of skills which crosses clerical and technical boundaries and a set of personal qualifications which enables the librarian to contend with the high rate of activity characteristic of a team. Through the life of the development there will be a need for the clerical services of typing, filing, and other business practices normal to an office environment. The demand for such services is heaviest at project beginning and end, and light in the middle. Between the start and end, different clerical skills which permit the librarian to maintain a set of computer files are required. To perform this service, the librarian needs to be skilled in the use of a keypunch or computer terminal and understand the procedures for preparing computer jobs to update machine readable files, for submitting those jobs for execution, and for assigning the results to their proper place.

To respond to the changing requirements over the life of a project, the programming librarian needs to be adaptive to change with emphasis on quick learning. Procedures are learned on the job. They need to be followed accurately and precisely, but they also need to be understood in relation to the major responsibility of recordkeeping. The librarian must be alert and responsive to the needs of the programmers especially since this position is the channel through which all development work flows. This job frequently requires physical activity and the librarian must be free of disabilities that preclude walking. This is especially true in a batch environment, less so in a terminal environment.

Corresponding to the chief programmer's leadership there is a characteristic feature of librarians that is important to the success of the team. That feature is a very positive attitude toward the job to be done. As with other team members, the librarian's personal characteristics reflect a stable personality willing and able to perform a challenging job.

EXERCISES

1. Explain the purpose of the CPT.

2. Name the three main members of the CPT.

3. List four duties required of the chief programmer.

4. Explain why the backup programmer position is necessary.

5. Explain the main function of the programming librarian.

6. Explain the importance of the programming librarian to the CPT.

7. Give an example of the duties of short term support members.

8. Explain the function of the project administrator.

9. List the qualifications of:

   a. Chief Programmer

b. Backup Programmer

c. Programming Librarian

10. What is the major concern of management in utilizing chief programmers?

384

372

CHAPTER 2

DEVELOPMENT SUPPORT LIBRARY

After studying this chapter, you should be able to:

1. State the purpose of a Development Support Library (DSL).
2. Give the principal objective of a DSL.
3. Explain why the DSL is composed of an internal library and an external library.
4. Explain the difference between a production library and a development library as they exist in a DSL.
5. Explain what is meant by the term "program stub."
6. Give the term for the automatic function of the DSL.
7. What member of the CPT works extensively with the DSL?
8. Explain how a program stub and a timing loop relate during the developing cycle.

DEVELOPMENT SUPPORT LIBRARY

The Development Support Library (DSL) serves as a central repository of all data relevant to the project, in both human readable and machine recognizable form. As such, it is used to organize and control the software development and is the focal point of information exchange - both management and technical - for the life of the project.

The principal objective of the library is to provide constantly up-to-date representations of the programs and test data in both computer and human readable forms. The DSL concept is designed to separate the clerical and developmental tasks of programming. In addition, the DSL makes the code produced more visible to the team members.

The components of a DSL, as an information base, are comprised of the internal and external libraries. The internal library consists of machine readable source programs, relocatable modules, object modules, linkage-editing statements, test data, or job control statements. The external library consists of all current listings of programs, as well as listings of recent versions of the programs.

In many projects a development support library is maintained by a librarian who interfaces directly with the computer. Programmers interface directly with the computer only on an exception basis. In order to permit this, a standard set of procedures (the computer or machine procedures) for performing all machine operations is required. These procedures contain all the necessary information for updating libraries, link-editing jobs and test runs, compiling modules and storing the object code, and backing up the libraries. By using these procedures, the librarian is able to perform any of the library operations without direct assistance. Other team members communicate with the librarian in such ways as submitting original coding sheets, making notations on directories, and indicating changes on source listings. (See figure 2-1.)

365

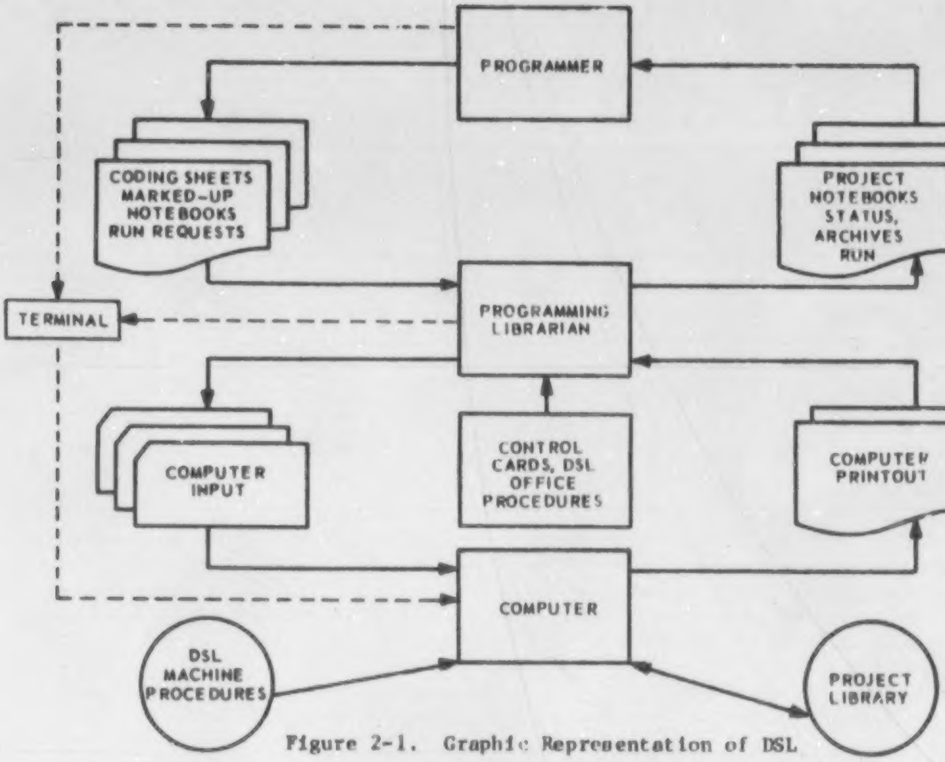


Figure 2-1. Graphic Representation of DSL

The DSL provides a significant aid for testing and evaluation in that the code is centralized to avoid ambiguity of what is, and what is not, valid software. A development support library will normally consist of a production library which contains code that has been tested and one or more development libraries for new code. At any point of the project, the overall production library constitutes the current operational system. Therefore considerable care is taken to see that new segments and data item definitions have been properly tested before they are added. This testing is performed in the development libraries where segments are created as needed and exist until the units have been tested and added to the production library. When a segment is added to the production library, it is removed from the development library. More leniency is allowed in adding to a development library than in adding to the production library. For example, if a segment references a data item for which it is not authorized, it cannot be added to the production library. Unauthorized access is permitted in a development library, although the user would be warned that he has committed an apparent error. Control is obtained by requiring that an update to the production library be conditioned on proof of successful testing in a development library. This will reduce the likelihood of errors getting into the system. The verification procedures are reviewed by the manager whose approval should be required for update to the production library.

The development support library provides the necessary control for programming of a system in a top-down manner. (See figure 2-2.) Testing and integration will start with the highest level system segment as soon as it is coded. Since this segment will normally invoke or include lower level segments, code must exist for the next lower level segment. This code, called a program stub, may immediately return control, may output a message for debugging purposes each time it is executed, or may provide a minimal subset of the functions required. These program stubs are later expanded into full functional segments, which in turn require lower level segments. Integration is, therefore, a continuous activity throughout the development process. During testing, the system executes the segments from the library that have been completed and uses the stubs where they have not. It is this characteristic of continuous integration that reduces the need for special test data drivers. The developing system itself can support testing because the code that interfaces with the newly added segments has previously been integrated and tested and can be used to feed test data to the new segments.

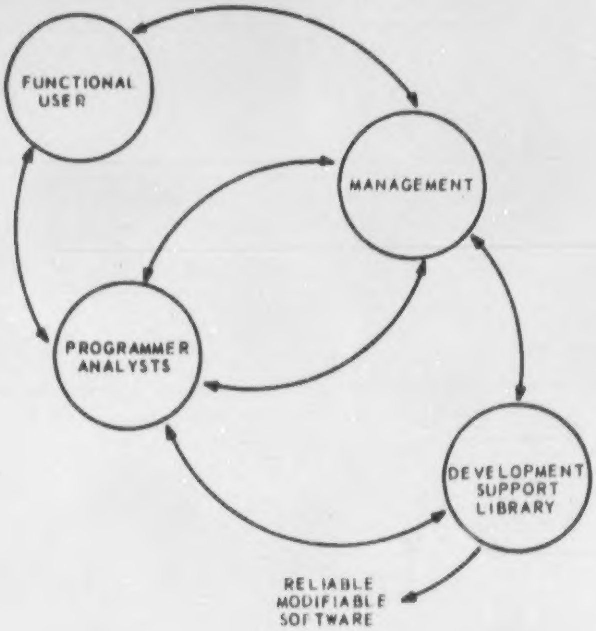


Figure 2-2. TDSP

Program stubs can often be created as an automatic function of the development support library. This automatic function is provided by the programming support library (PSL). The PSL is a software system which provides the tools to organize, implement, and control computer program development. The system is designed specifically to support top-down development and structured programming. Different implementations of a PSL exist for various computer and operating system environments used in system development. The fundamental correspondence between the internal and external libraries in each environment is established by the PSL office and computer procedures. The office procedures are specified at a detailed level so that the format of the external libraries will be standard across programming projects, and the maintenance of both internal and external libraries can be accomplished as clerical functions. The PSL computer procedures for each are expressly designed for easy invocation by librarian personnel so that their use is nearly fail-safe.

The use of the top-down approach with a library provides a basis for capturing performance data during the development cycle. By replacing each stub with a timing loop that utilizes the estimated run time for that function, the developing system becomes a model. As dummy routines are replaced with working code, the performance results can be appraised against the performance objectives. In a similar manner, storage allocation can be modeled.

The use of a development support library combined with structured programming, top-down development, and documentation significantly improves management control of the software development effort by providing continuous product visibility. Since the developing system is undergoing continuous integration, the system status is accurately reflected through the contents of the library; i.e., completeness is measured objectively in terms of how much of the system is operational. The completed code can be reviewed to verify status and appraise the quality of the software product.

EXERCISES

1. State the purpose of a Development Support Library (DSL).

2. Give the principal purpose of a DSL.

3. Explain why the DSL is composed of an internal library and an external library.

4. Explain the difference between a production library and a development library as they exist in a DSL.

5. Explain what is meant by the term "program stub."

6. Give the term for the automatic function of the DSL.

7. What member of the CPT works extensively with the DSL?

8. Explain how a program stub and a timing loop relate during the developing cycle.

CHAPTER 3

STRUCTURED WALK-THROUGHS

Upon completion of this chapter, you should be able to:

1. State why the structured walk-through concept was developed.

2. Discuss how structured walk-throughs will relate to the chief programmer teams and development support library concepts.

3. Explain how the structured walk-through will operate.

4. Give the main argument against structured walk-throughs.

5. Describe management's role in the structured walk-through.

Management realized that the Chief Programmer Teams needed some form of scheduled reviews to determine project status at any given point. The DSL as discussed in the previous chapter would greatly enhance these reviews by providing "constantly up-to-date representations of the programs and test data." Recognizing this need, IBM developed the concept of structured walk-throughs as part of its chief programmer team approach to project organization. Other organizations have referred to the same concept as "team debugging" and a variety of other terms. Structured walk-throughs can be thought of as a set of formal procedures for reviews - by the entire programming team - of program specifications, program design, actual code, and adequacy of testing. If implemented properly, structured walk-throughs go hand in hand with the concepts discussed in the previous two chapters as well as the concepts to be introduced in Section II.

Since the advent of the computer age, programmers have been urged to perform extensive desk-checking before running their programs on the computer; however, desk-checking began to be enforced less and less as scarce computer time became more and more plentiful in the late 60's. In some organizations, the advent of time-sharing virtually eliminated the concept of desk-checking altogether. Structured walk-throughs are, in a sense, a return to the old philosophy of desk-checking, but with an important difference. Now we are suggesting that a group of programmers should review the design and coding of an individual programmer.

The real importance of the structured walk-through concept lies in its connection to top-down design and coding. At a very early stage in a programming project, the programmer should be able to show to other members of his team the design of the entire program. Much of the program will be in the form of program stubs (dummy modules) but the overall logic and structure should be present. The goal of the team review is to insure that the overall logic of the program is correct. Rather than worrying about the details of low-level modules, the team concentrates its efforts, at the beginning of the project, on a review of the high-level structure of the program, thus identifying any major flaws that the programmer may have overlooked. Structured walk-throughs are technically sound but the human element will generate some "resistance to change."

Upon being introduced to the concept of structured walk-throughs, many programmers react negatively. They feel that it will be too expensive to have four or five programmers reviewing the work of one programmer and that the project will be greatly impeded by work stoppages to conduct the walk-through. Once using the structured walk-through approach, programmers will realize that it saves a considerable amount of time. First of all the walk-through only requires a couple of hours to review a week's work. Second, by having four or five programmers look at your design, with varied viewpoints, the chances are very good that a majority of the bugs will be eliminated before the program is ever run.

Edward Yourdon in his book, "Top-Down Program Design," states that it seems that the real objection that many programmers have to the structured walk-through approach is that they prefer not to have other people looking at their code. Many programmers find the structured walk-through an ego-bruising experience. But this should not be the case, as Gerald Weinberg explained the "egoless programming" philosophy in his book, "The Psychology of Computer Programming." Personality conflicts can completely destroy the entire project. These conflicts must be eliminated or at least minimized for the structured walk-through to be successful.

Within IBM the structured walk-through is:

1. A positive motivation for the project team.
2. A learning experience for the team.
3. A tool for analyzing the functional design of a system.
4. A tool for uncovering logic errors in program design.
5. A tool for eliminating coding errors before they enter the system.
6. A framework for implementing a testing strategy in parallel with development.
7. A measure of completeness.

Also within IBM the basic characteristics of the structured walk-through are:

1. It is arranged and scheduled by the developer (reviewee) of the work product being reviewed.

2. Management does not attend the walk-through and it is not used as a basis for employee evaluation.

3. The participants (reviewers) are given the review materials prior to the walk-through and are expected to be familiar with them.

4. The walk-through is structured in the sense that all attendees know what is to be accomplished and what role they are to play.

5. The emphasis is on error detection rather than error correction.

6. All technical members of the project team, from most senior to most junior, have their work product reviewed.

Experience with structured walk-throughs has been most encouraging. Undoubtedly, there are a number of ways they could be modified to fit other organizations. The central idea, however, should remain the same; i.e., to convert the classical project review into a productive working session which not only tracks progress but which makes a positive contribution to that progress. Outwardly, management involvement appears low, but in reality structured walk-throughs provide management with a vehicle for catching errors in the system at the earliest possible time when the cost of correcting them is lowest and their impact is smallest.

EXERCISES

1. State why the structured walk-through concept was developed.

2. Discuss how structured walk-throughs will relate to the chief programmer teams and development support library concepts.

3. Explain how the structured walk-through will operate.

4. Give the main argument against structured walk-throughs.

5. Describe management's role in the structured walk-through.

SECTION II

TDSP DESIGN TOOLS AND TECHNIQUES/DOCUMENTATION

INTRODUCTION

Section II discusses the various design aids and design techniques used in Top-Down Structured Programming. The documentation tools are presented first, beginning with Data Flow Graphs (Chapter 4). Chapter 5 explains the many facets of structure charts. HIPOs (Hierarchy plus Input-Process-Output) are taken up in Chapter 6. Structured flowcharting (Nassi-Schneiderman diagrams) is the topic of Chapter 7. The various techniques of structured design are described in Chapter 8. Chapter 8 begins with Transform Analysis, with module coupling and module cohesion following. Transaction Analysis concludes the chapter. Each technique explores the problem from a different avenue and provides a different graphical representation of the problem. There is no best tool or technique, since they all provide valuable information to the design process.

To gain a full understanding of the material in this section, it may require reading the Section twice; first for familiarity, then secondly for comprehension.

CHAPTER 4

DATA FLOW GRAPHS

Study of this chapter should enable the student to:

1. Define the function of the data flow graph.

2. Explain at least three of the benefits that may be derived from use of the data flow graph.

3. Construct a data flow graph as specified in a simple problem statement.

The Data Flow Graph (DFG) is a simple yet effective method of organizing and recording the initial ideas for the solution to a problem. Like a "system flowchart," the emphasis is on the flow of data through the problem. The solution should correspond to this data flow; that is, for each piece of the problem, there is a corresponding piece of the solution.

There are several benefits obtained by taking the time to construct the data flow graph(s):

1. A data flow graph becomes a starting point in the solution process. Rather than getting tangled in details, it is easier to provide an overview of the problem to lead to the design of the solution.

2. There is also less of a chance that time will be wasted in solving the wrong problem. How does this happen? By ambiguous specifications or invalid assumptions. Often the user is not sure of what he wants. By providing a data flow graph early in the design stage, the problem stands a better chance of being understood by everyone involved.

3. Then the data flow graph becomes a communication document for walk-throughs by managers and users, or by other programmers. Even if there is not a formal walk-through, a data flow graph can be a point of discussion to anyone related with the problem.

4. As a form of documentation, a data flow graph provides the maintenance programmer with a simple overview of the problem. A data flow graph is much easier to understand and relate to the overall organization of the solution, than deciphering code to obtain the same picture.

The data flow graph uses simple graphic elements to show the progression of data items from input to output. Decisions and control logic are not shown, only the major data items and their processes.

The elements of the data flow graph are:

⟶	The arrow shows the flow of data through the graph. The arrow will be labeled with the associated data item identifier.
○	The circle indicates a process being performed. A name, descriptive of this process, would be written in the circle (it is often easier, for size considerations, to name the process and then draw the circle). Due to the graphics, a data flow graph is sometimes referred to as a bubble chart.

Figure 4-1 shows that Data A is used by process B producing Data C. Data C is used by process D which then produces Data E. In this example, A is the initial input and E is the final output. C is an intermediate data item.

It is important to correctly describe the data as it is processed in a data flow graph.

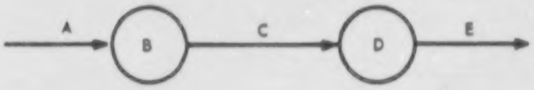


Figure 4-1

Figure 4-2 shows the use of abstract data items and the general processes necessary to change the data from one form to the next.

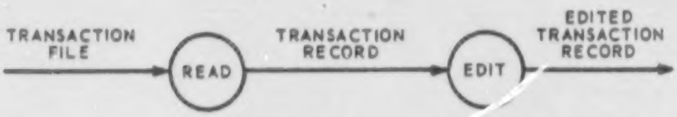


Figure 4-2

Suppose the problem statement said: "Provide edited records from the transaction file."

The statement is ambiguous, to say the least. Still, figure 4-2 represents the way one person interpreted the problem. An assumption was made that a part of the problem is to obtain a record from the transaction file. The problem could be further interpreted into more detail by making more assumptions. However, this could be a wasted effort.

Consider this problem: Update the master tape file.

The solution:

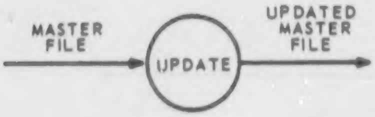


Figure 4-3

And if some assumptions are made:

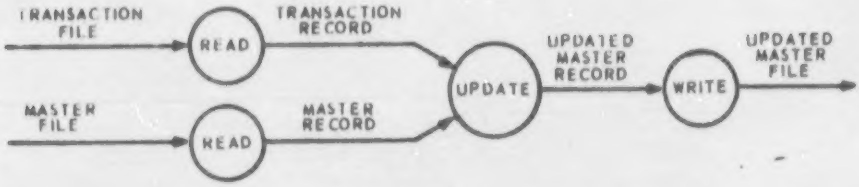


Figure 4-4

It is obvious that more information is needed or a great deal of time could easily be wasted solving the wrong problem. Rather than complain that the problem is ambiguous, a data flow graph can be presented for elaboration and constructive comment. The user can get an idea of what the programmer is doing. Both can effectively communicate.

The data items of the data flow graph are not computer oriented. They are in human terms and depend on the interpretation of the problem for their description. The data items are carefully labeled to show each step in the transformation process. The inability to visualize the data flow indicates a lack of understanding of the problem.

If the problem for figures 4-3 and 4-4 was further described to include sequence checking, the various transaction edits, and the requirement to process all the information for a particular record, then the next more detailed data flow graph might look like figure 4-5.

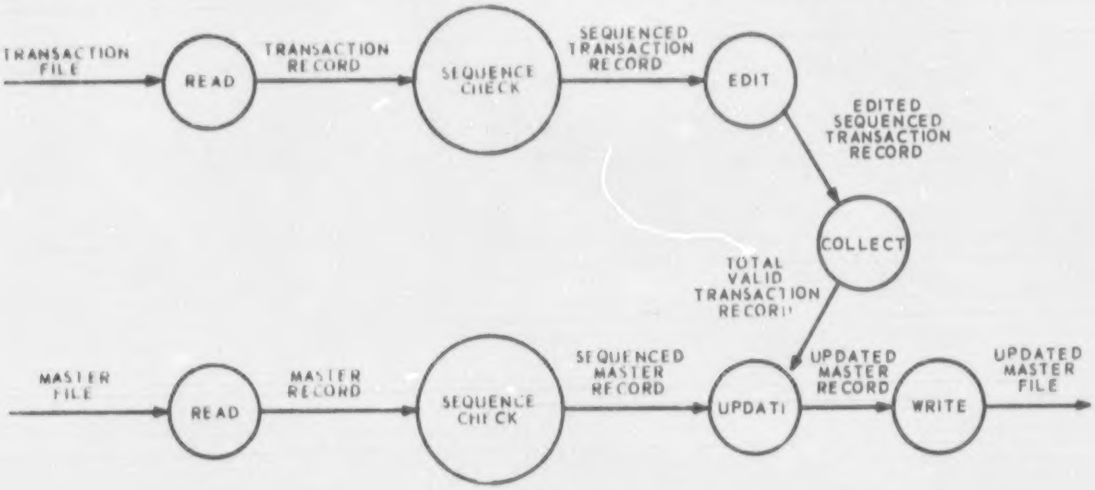


Figure 4-5

The data flow graph in figure 4-5 does not show the control logic and decisions. It is not the purpose of the data flow graph to solve the problem - only to define it. Notice that the size of the "bubble" does not correspond to the size or complexity of that particular part of the problem. The graph is not concerned with how the data flow is initialized or terminated nor how subsequent records are processed. There are no flags, switches, or extraneous data items which will probably be required in the solution. There are no error paths, although it can be assumed that there could be errors, since the problem requires sequence checks and edits.

All of these details are left to the lower levels of design where the question of how to solve the problem is answered. The data flow graph is telling what the problem is.

Examples have been shown using a data flow graph with vague and brief problem statements. If the problem specifications run into some number of pages or even a book, a data flow graph is still a valid tool to organize the thought process for visualizing the overall problem. If the problem is very complex, it may be easier to understand if it is examined with several levels of data flow graphs.

In constructing a data flow graph, it may be easier to start with the output(s) and work back toward the inputs. There are no absolute rules for constructing data flow graphs, but some guidelines may be helpful.

1. Do not show control logic or decisions.

2. Ignore initialization and termination. Pretend that it all "runs" at once and stays running.

3. Label transitions very carefully as to data passed.

4. Make sure that data flow is correct for the level of detail being shown.

5. Do not be overly imaginative. Express the problem as clearly and simply as possible.

6. Identify assumptions for further clarification before further design.

7. Omit simple error paths from each "bubble" to the outside.

8. Work from input to output, or vice versa, until stuck - then switch.

9. Don't flowchart.

A data flow graph does not produce a program. It only illustrates the way an individual interprets the problem. The data flow graph is used for further design strategy to provide clues to a good design organization. These strategies will be discussed in Chapter 8.

EXERCISES

1. What is the purpose of the data flow graph?

2. Give three benefits of data flow graphs.

3. Construct a data flow graph for the following problem:

   "Edit the master file to delete records which are outdated. The new file will be sorted."

CHAPTER 5

STRUCTURE CHARTS

Upon completion of this chapter, the student will be able to:

1. Identify and explain the use of the graphic elements of structure charts.
2. Explain basic facts relating to the use of structure charts.

The structure chart is a design tool which is noted for its simplicity and flexibility. It is used to organize and document the thought process leading to the solution of a problem. The structure chart can be used at any level of design to break the solution into simpler components. Simple graphic elements allow several design concepts to be evaluated with minimum effort.

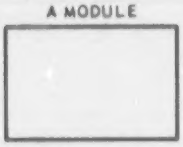


Figure 5-1

Figure 5-1 represents a functional part of the solution. For convention, this functional part will be referred to as a module, since it will be considered "an assembly, functioning as a component of a larger part." A brief description (usually one or two words) of the function of the module is placed within the figure.

Figure 5-1 is used for modules which are yet to be developed. To illustrate a module which is predefined (such as a library subroutine), figure 5-2 is used.

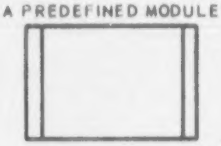


Figure 5-2

The design, using a structure chart, is composed of these modules, interconnected to show their relationship or hierarchy to one another.

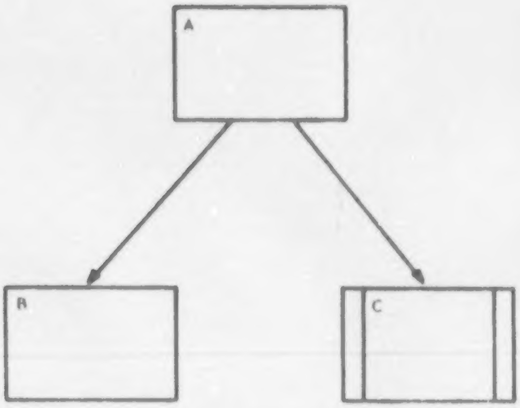


Figure 5-3

An example of this relationship is shown in figure 5-3, where module A controls B and C (B and C are subordinate to A). In other words, for A to perform its function, it must perform B and/or C in some combination. The control of B and C is in module A. Somewhere in A, a normal transfer is made to execute B. After B has performed its function, control is returned to the point in A where the transfer originated and the next statement is executed. This is described as a normal connection and is read as "A calls B." Also somewhere in A is a reference to module C which is a predefined module.

This structure chart does not show logic. There are no clues as to the order of execution of modules B and C, the number of times they are called, or if one or both modules are required each time A processes data. Suppose module C was a library sort routine and module B was an error handling function. Most of the time module B would not be needed, but the error possibility does exist in the problem, so B is there when it is needed. Module A determines when this need occurs.

The rules of structure charts allow wide variations in the graphic arrangement of modules, as shown in figure 5-4. For this reason, the lines between modules have an arrowhead to show control. Both examples illustrate the same concept. Module A controls modules B and C.

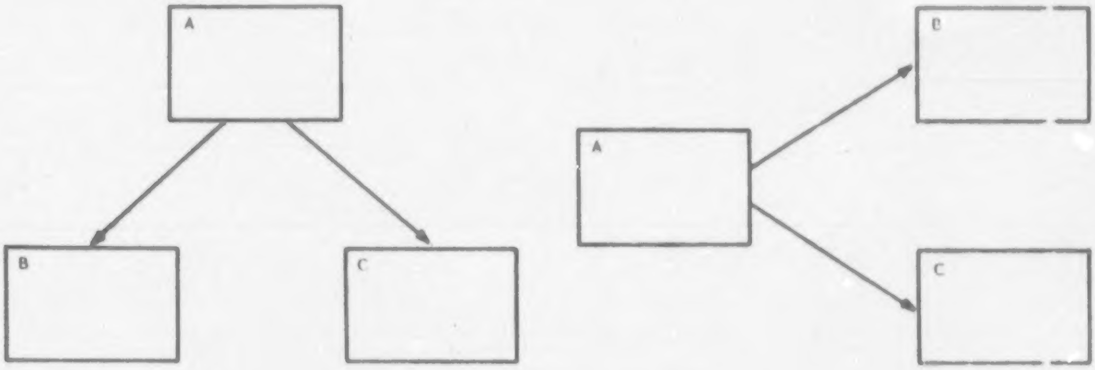


Figure 5-4

387

So far, all the references between modules have been done directly using normal connections as in figure 5-5. Notice that the line for this normal connection does not enter A or B since the line would then indicate internal referencing which is explained later in this chapter. There is only one line from A to B, even though there may be several distinct calls to B from A.

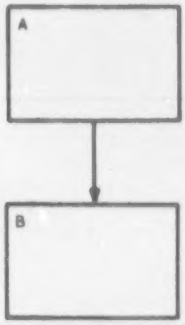


Figure 5-5

The structure chart allows the structure or organization of the design to be easily shown. Figure 5-6 illustrates a basic structure chart. Module E is invoked by two modules. This structure may cause modification problems since a change in E might now impact both B and C.

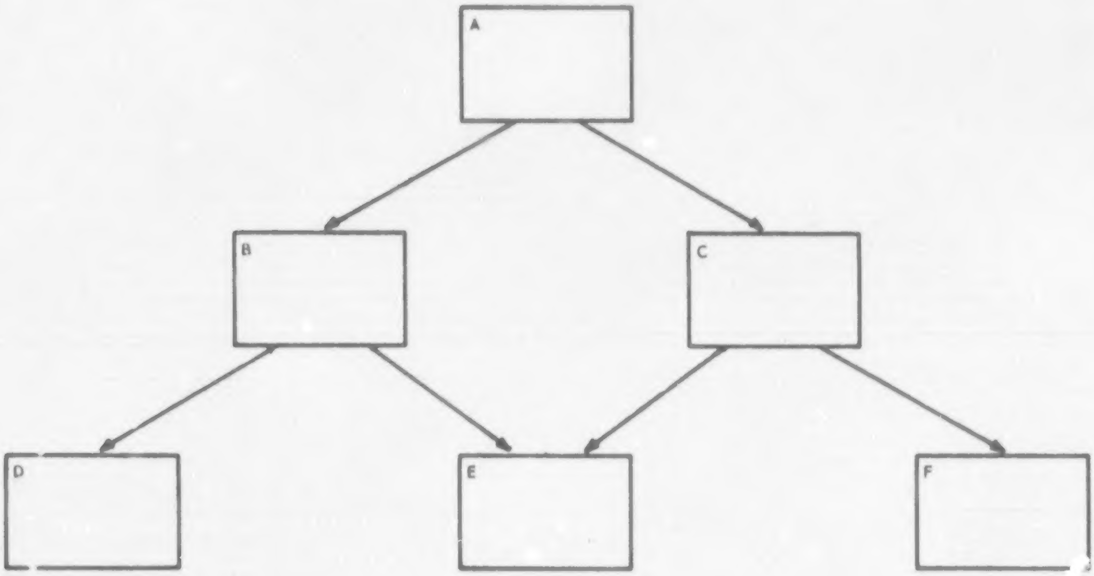


Figure 5-6

If this is considered to be a problem, then the design can be reorganized so that B and C can control separate E modules as in figure 5-7. The point to be made is that although figure 5-6 appears to be more efficient and requires less work, it is also more complex, making it more difficult to debug and modify. Efficiency is a two-sided coin. The time spent considering the design will be more than repaid over the life of the software product. Deciding the best way to organize the solution is the real work of design.

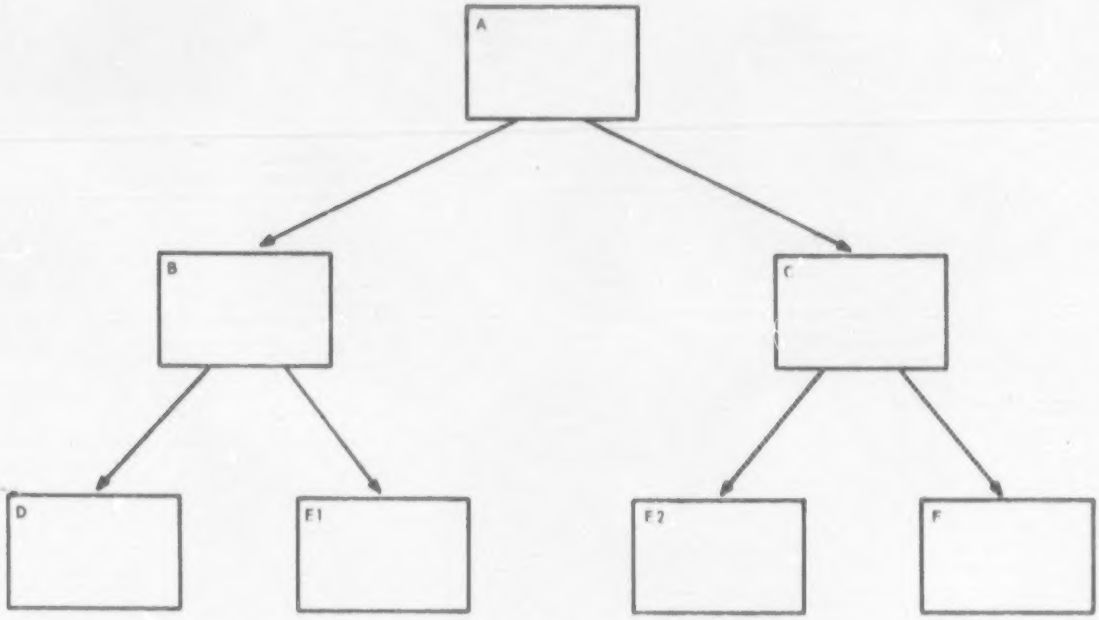


Figure 5-7

Structure charts seldom show logic. Major decisions and iterations may be shown, but they are not widely used because they clutter up the design and have little meaning. Figure 5-8 shows the limited logic symbols for structure charts. Module A invokes module B based upon some decision. Module A also contains a loop which calls for module C to execute some number of times.

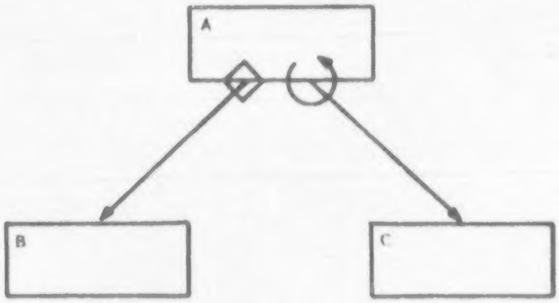
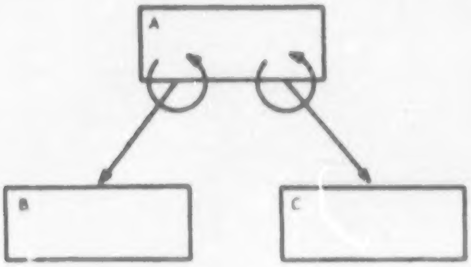
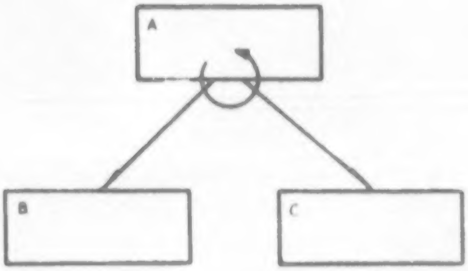


Figure 5-8

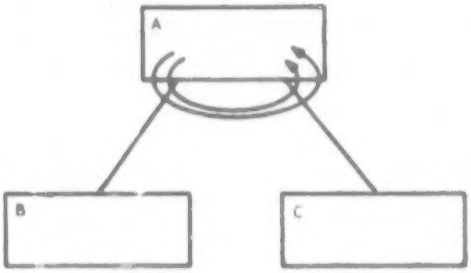
Figure 5-9 shows some examples of structure chart logic at the program level and the coding that might be expected in each of the A modules.



```
          .                    .
          .                    .
          .                    .
    DO 5 I = 1,N          PERFORM B N TIMES
    CALL B                     .
  5 CONTINUE                   .
          .                    .
          .               PERFORM C M TIMES
          .                    .
    DO 10 I = 1,M              .
    CALL C                     .
 10 CONTINUE
          .
          .
          .
```



```
          .                    .
          .                    .
          .                    .
    DO 5 I = 1,N          PERFORM B THRU C
    CALL B                        N TIMES
    CALL C                     .
  5 CONTINUE                   .
          .                    .
          .
          .
```



```
          .
          .
          .
    DO 5 I = 1,N
    DO 10 I = 1,M
    Call B
    Call C
 10 CONTINUE
  5 CONTINUE
          .
          .
          .
```

Figure 5-9

All discussion thus far has centered around the normal connection between modules. There may be a situation requiring another type of control organization in a structure chart. Figure 5-10 illustrates the graphics of this "pathological control" connection. All modules up to this point have maintained a one-in one-out concept. That is, if a module was initiated, it did so at a single entry point and terminated at a single exit point. This was done regardless of references or calls to other modules. The control always returned to the point where a module was invoked. The pathological control connection is an abnormal exit from or entrance to a module. An example can be made from figure 5-10. Suppose a requirement existed to edit a field, checking for a specific range. If the field was above the range, module A would do one thing; if it were below, A would do something else. Although this could be handled with a normal connection by checking the results of B upon the return to A, suppose B returned control to the corresponding part in A thus eliminating the check in A.

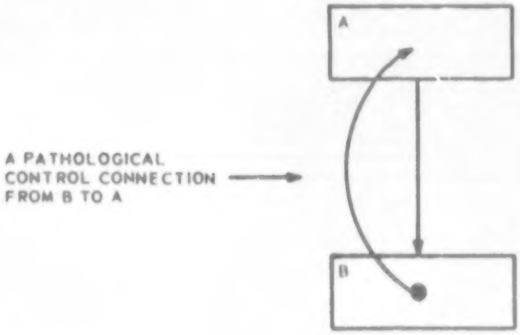


Figure 5-10

The pathological control connection is used if the cost of a normal connection is too high. The associated cost factors should include programmer efficiency as well as machine efficiency. The latter has unfortunately been given too much emphasis resulting in software products which are difficult to develop, document, debug, and modify. The pathological control connection generally increases the complexity of the solution to such a degree that many authorities on the subject strongly advocate not using this abnormal connection.

Besides showing the control structure or hierarchy of modules, a structure chart also shows the communication between modules by indicating data flow. Just like connections, there are two general categories of data flow, normal (direct) and pathological (indirect). The distinction between the two can be labeled direct and indirect.

The normal data flow (direct) is illustrated in figure 5-11. Module A passes data information "X" to module B, and B returns control information named "Y" to A. Notice that ○⟶ is used for data and ●⟶ means control. The distinction between "data" and "control" type information is determined by the primary purpose of the information item. To add clarity to the software effort, each information item should be used for a specific purpose. This rule is often violated when the need for a control flag arises and an "empty" information item (one which has served its purpose and is no longer needed) is used for this additional duty. Again the question of efficiency must be answered. Is the space (core) which is saved worth the increase in complexity which results in documentation, debugging, and modification problems?

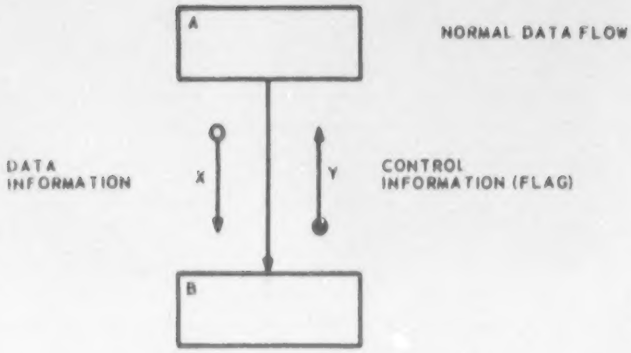


Figure 5-11

The concept of data flow between modules is difficult to define, but it is important. This concept should not be thought of as being implemented by a particular language. It exists at the more abstract design level and once data flow has been designed into the system, the programming follows with the same data flow complexity as the design. In COBOL, the data flow between modules (paragraphs) must be controlled by the design since all data items are declared and are accessible by all the modules (paragraphs) in the program. The ideal condition would be to give a module only those information items which it requires to perform its function. This is a management and security concept of limiting information to a need-to-know basis.

FORTRAN subroutines allow restricted access to data on a need-to-know basis since the data items are passed as parameters to and from the subroutine. Although there is no distinction between parameters which are input or output by the calling statement to a subroutine, the data flow is still established by the design and reflected by the program. Some programmers feel that data flow is easier to handle if all modules have access to all data items. Every data item is passed to each module either through the call or via named or blank common. The results can be disastrous over the life of the system or program when the organization of the data is misunderstood and changed for a modification or for efficiency.

A structure chart which passes a large number of data items may indicate a poor design and, therefore, require a redesign of the system or program. If it is determined that the design is good, then a cross-referencing scheme (using the same or separate pages) can be used to alleviate the clutter caused by using meaningful data names within the confines of a structure chart.

The other category of data flow is the pathological data connection shown in figure 5-12. The graphics deviate to illustrate this concept in a structure chart since the data "flow" is against the arrow. Module A has a normal control connection to B and there is also an indirect reference to data item X which exists in B. The reference is from A to B, but the data flows against the arrow from B to A. The best example of this type of connection is the COMMON declaration in FORTRAN. If the main program and subroutines are thought of as modules, then data passage can be done directly by listing the parameters to be passed, or indirectly by referencing data in COMMON. Although the latter seems easier and more efficient, it would be wise to consider the problems of debugging and modifying such a connection as compared to explicitly listing the data items passed.

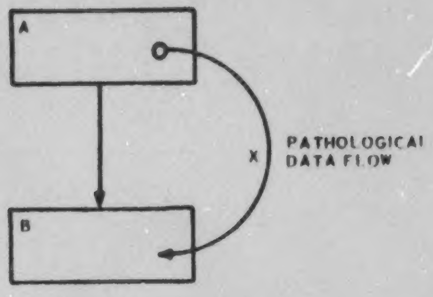
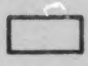
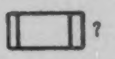


Figure 5-12

The mechanics of structure charts are simple and flexible. Several designs can be considered with minimum effort. The initial organization of the structure chart is important since it is the basis for all the design work to follow, so some extra time should be spent analyzing this initial design to achieve the best possible system. Chapter 8 discusses several design considerations to provide clues as to how to organize a good structure chart. A programmer must use structure charts several times before gaining the desired proficiency and the full appreciation of same.

STRUCTURE CHARTS

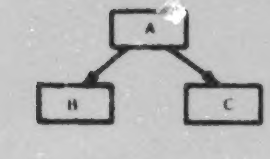
1. What is a module?

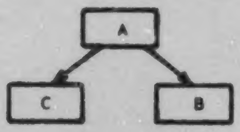
2. What is represented by  and ?

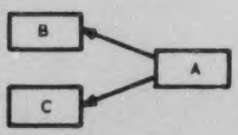
3. How is a normal connection between modules represented?

4. What is the purpose of a structure chart?

5. What is the difference between







6. What do ◇ and ↻ represent?

7. What is a pathological control connection? Why should it be avoided?

8. What symbol is used to represent passing data information? Control information?

9. What can be done to reduce data items being passed?

10. Why is the initial structure chart important?

394

CHAPTER 6

HIPOs

Upon completion of this chapter, the student should be able to:

1. Define the acronym "HIPO."

2. List the three diagrams used in the HIPO package and explain the use of each diagram.

3. Explain the purpose of graphics in HIPO construction.

4. Explain when HIPOs are initiated in the system development.

5. Explain how HIPOs are used at each stage in the development.

INTRODUCTION

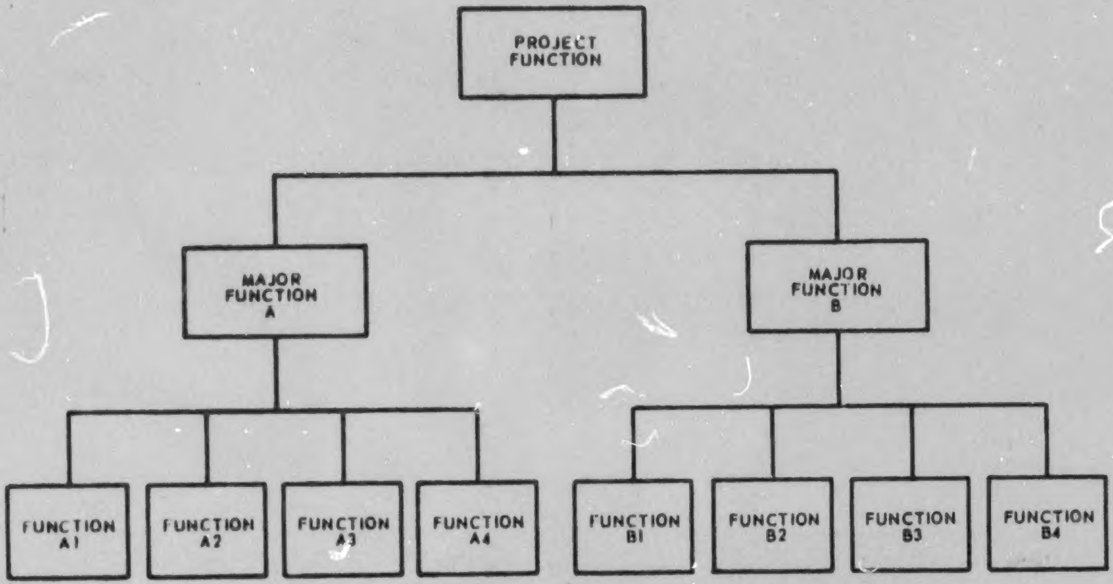
IBM developed a design and documentation tool to complement the top-down design, structured programming, and chief programmer team methodologies. The tool designed was hierarchy plus input-process-output (HIPO). HIPO aids in system development by providing a graphical representation from the initial design through the life of the system. The HIPO package consists of three diagrams which provide a complete pictorial and verbal representation of the project. The HIPO approach to system design is from the general to the specific, as the top-down design philosophy is to system development. Flowcharting is a useful tool but is aimed at the logical structure of a program. HIPOs, on the other hand, are concerned with the functional development of the system. HIPO allows for a formalization of the designer's thoughts at an early stage of the development. It is a formal guide to programmers during the coding phase. HIPOs continue to be used during system maintenance. HIPOs are used by management and user groups during system reviews. It provides a documentation and design tool for all corporate levels from management to the programmer/coder. The following sections include the diagrams which compose the HIPO package, the use of graphics in HIPOs, and how HIPOs should be used.

HIPO DESCRIPTION

The HIPO package consists of overview diagrams, detail diagrams, and Visual Table of Contents (VTOC) also known as hierarchy charts. Each diagram provides a level of definition in the design of the system. The VTOC and overview diagrams, which aid the early design, can both be worked on simultaneously and usually complement each other. The detail diagram refines the system design with specific details for particular functions.

The VTOC represents the design in a hierarchal structure as shown in figure 6-1.

387



THE PROJECT PLAN

Figure 6-1

The names and identification of all overview and detail diagrams appear in the VTOC. The names refer to a verbal description of each box in the structure. The identification is a suitable numbering system for the hierarchy chart which is used as a cross reference with the other diagrams in the HIPO package. There are many numbering systems that can be used. Figure 6-2 demonstrates one possible system.

It may be necessary to continue the hierarchy chart on other pages depending on the size of the system and the design. It is only necessary to copy the block of the diagram to be continued as shown in figure 6-3.

Although graphics will be discussed later, it is important to note that the first page of the VTOC should provide a legend of symbols used in the HIPO package. This provides a common ground for those individuals using the package.

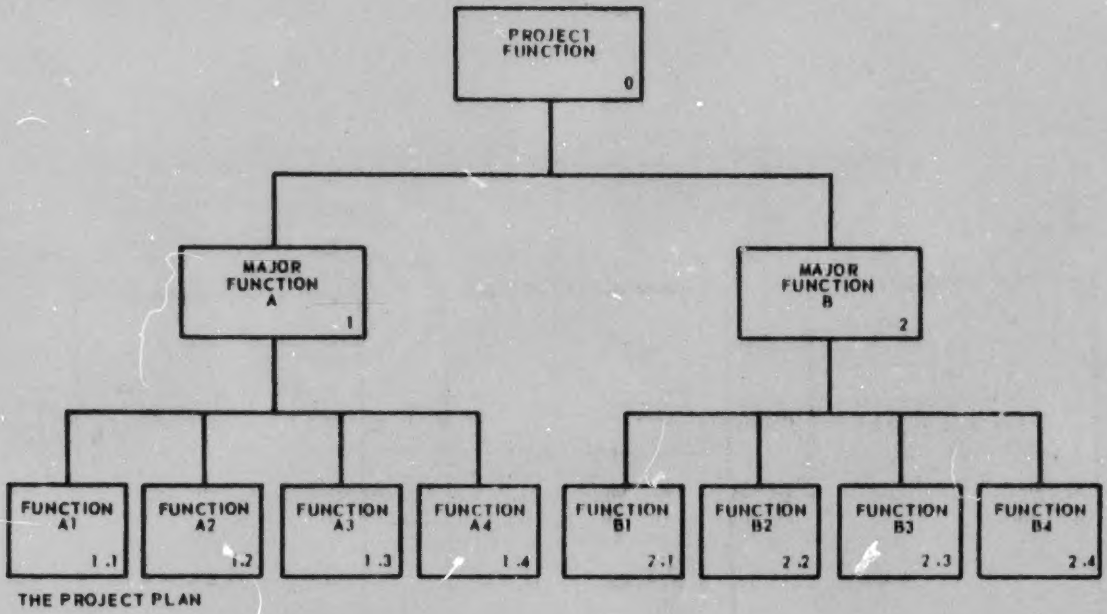


Figure 6-2

The overview and detail diagrams are composed of three sections which supply the inputs required, the processes needed, and the outputs produced for the function being described. Each diagram has an extended description section for further explanation when clarity of the diagram is necessary.

The overview diagram presents a general description of a function and usually supplements the hierarchy chart (also known as the VTOC) in the early stage of the design. It refers to all detail diagrams which add definition to the problem being discussed. Overview diagrams list inputs, processes, and outputs in a general fashion and make no attempt to relate them to each other. Figure 6-4 is an example of an overview diagram.

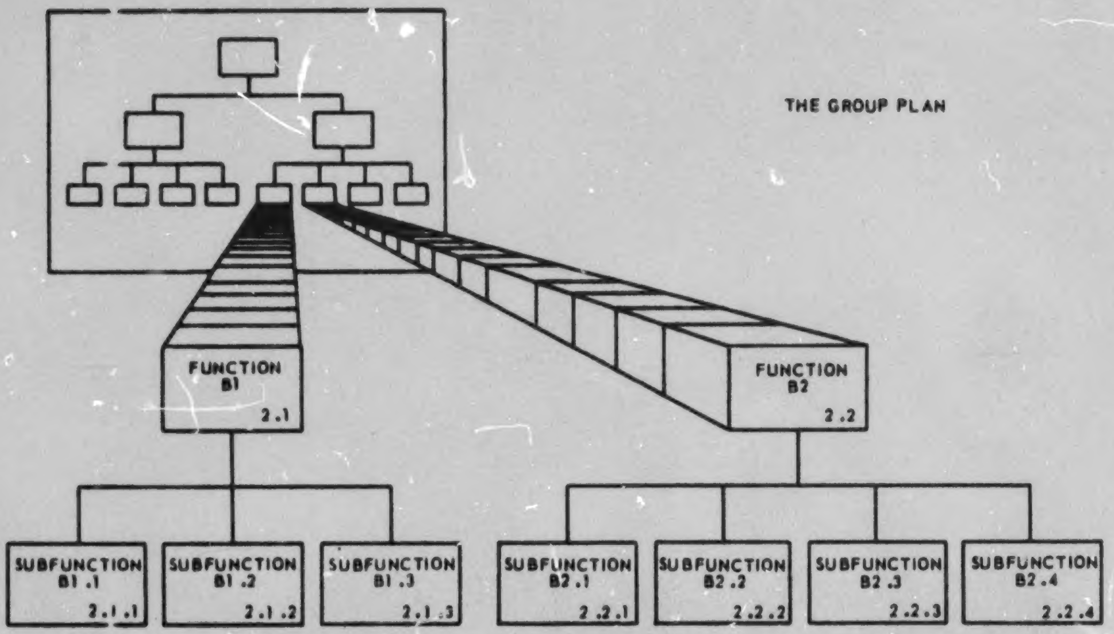


Figure 6-3

The detail diagrams provide specific information at the lower levels of the hierarchy chart. The complexity of material, the number of function subprocesses, and the amount of documentation determine the level of detail needed. As a first step in the detail diagrams, it is possible to take the overview diagram and connect the inputs and outputs to the appropriate processes as shown in figure 6-5.

It is also helpful to combine and consolidate steps in the diagram, thus reducing the number of connections. Rearranging the inputs, processes, and outputs can also alleviate the amount of line crossing in the diagram. Figure 6-6 demonstrates combining, consolidating, and rearranging steps. It is important to note the use of graphics.

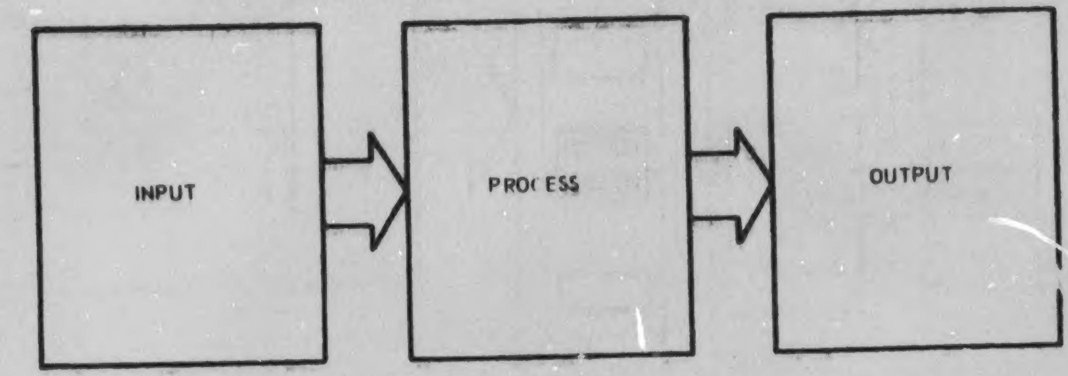
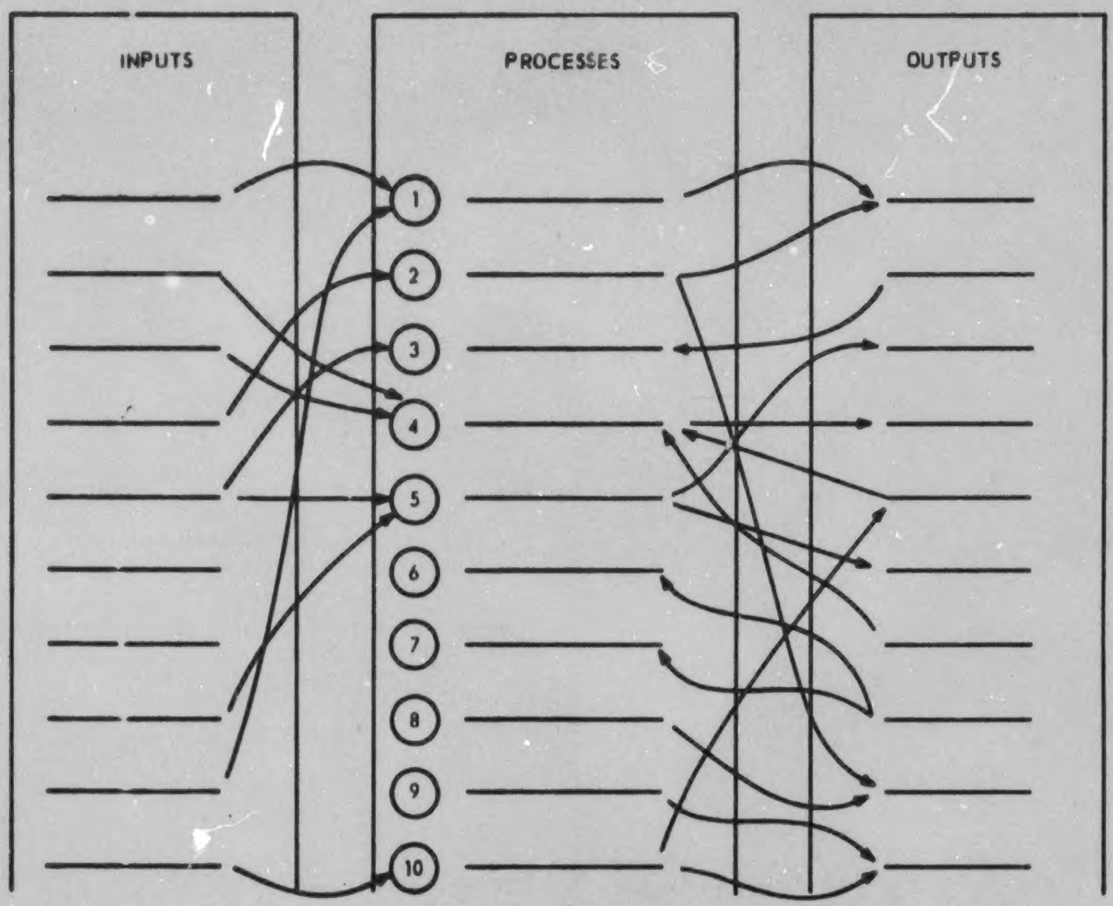


Figure 6-4

Figure 6-7 illustrates the detail diagram with the use of the extended description and a complete set of graphics.

Graphics

In the original conception of HIPO, the IBM organization developed an extensive set of graphic symbols to aid in the clarity of the HIPO diagrams. The degree to which graphics are used is dependent on the using organization. Graphics should complement the diagrams so far as to remove ambiguity. Graphic standardization, within the organization, is necessary to eliminate user misconceptions. It is recommended that a reasonable subset of the graphic symbols be incorporated to provide a clear, concise, and complete HIPO package. The following figures serve to illustrate some preferred methods of graphic use.



DRAW 3 OPEN-ENDED BOXES

LIST INPUTS, PROCESSES, OUTPUTS

CONNECT THEM

BUT DON'T WORRY ABOUT GRAPHICS

MAKING A HIPO DIAGRAM-II

Figure 6-5

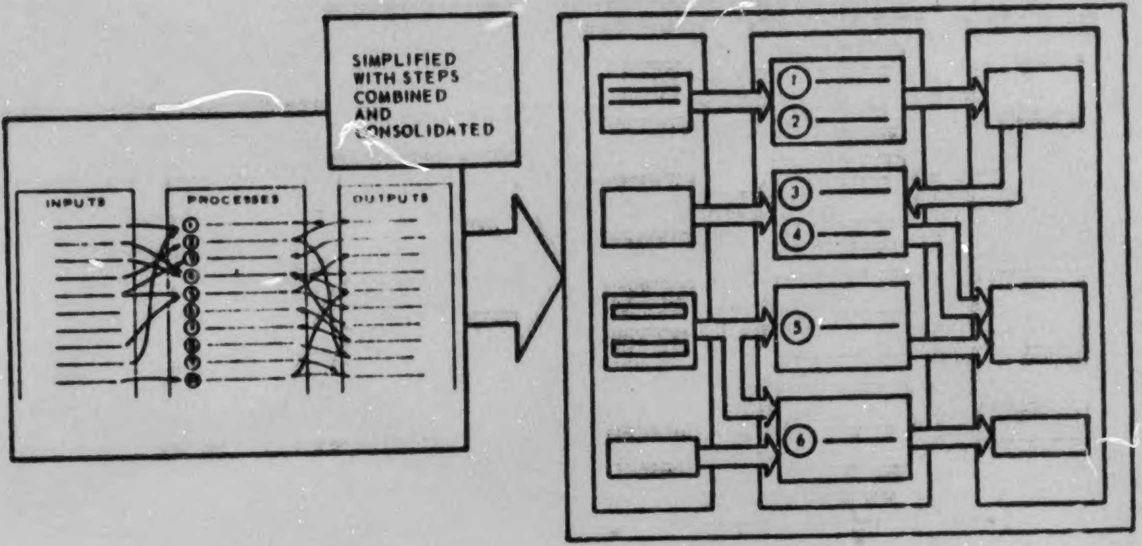
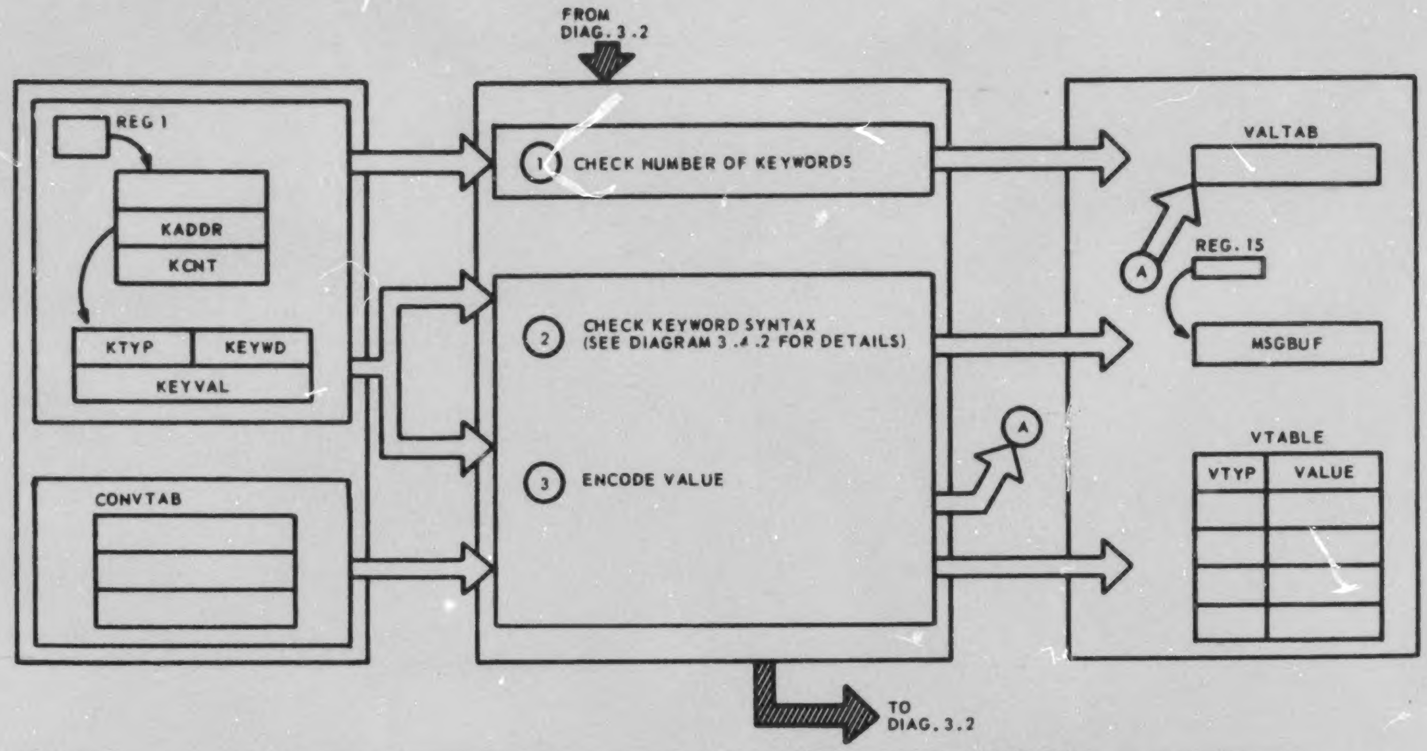


Figure 6-6

Grouping with boxes normally shows relationships while arrows normally show movement. It might be necessary to distinguish between data and control movement. Arrow graphics can be a time-consuming process. Figure 6-8 depicts one method of representing arrows, but it should be noted that single lines will accomplish the same results.



LEGEND	
➡	ENTRY TO DIAGRAM EXIT FROM DIAGRAM
⇨	DATA MOVEMENT
⟶	POINTERS
- - - -➝	REFERENCES

STEP	DESCRIPTION	CROSS REFERENCES
①	LIMIT OF 15 SEEMS REASONABLE. PUTTING A COUNT SOMEWHERE WOULD BE HELPFUL – KEYTAB	SEE DES. OBJ. PP 37–42 SEE PLM PP 104–112
②	USE EXISTING SYNTAX CHECKER AEQ4107A BUT MODIFY TO USE/AS DELIMITER	SEE PLM PP 18–20 F.C. NO. 12
③	PERFORM BINARY SEARCH OF CONTAB TO FIND CONVERSION ALGORITHM. CONVERT VALUE AND STORE IN NEXT AVAILABLE FIELD OF VTABLE WITH TYPE INDICATOR	MODULE AEP3204B SPEC PP 37, 43, 82

Figure 6-7

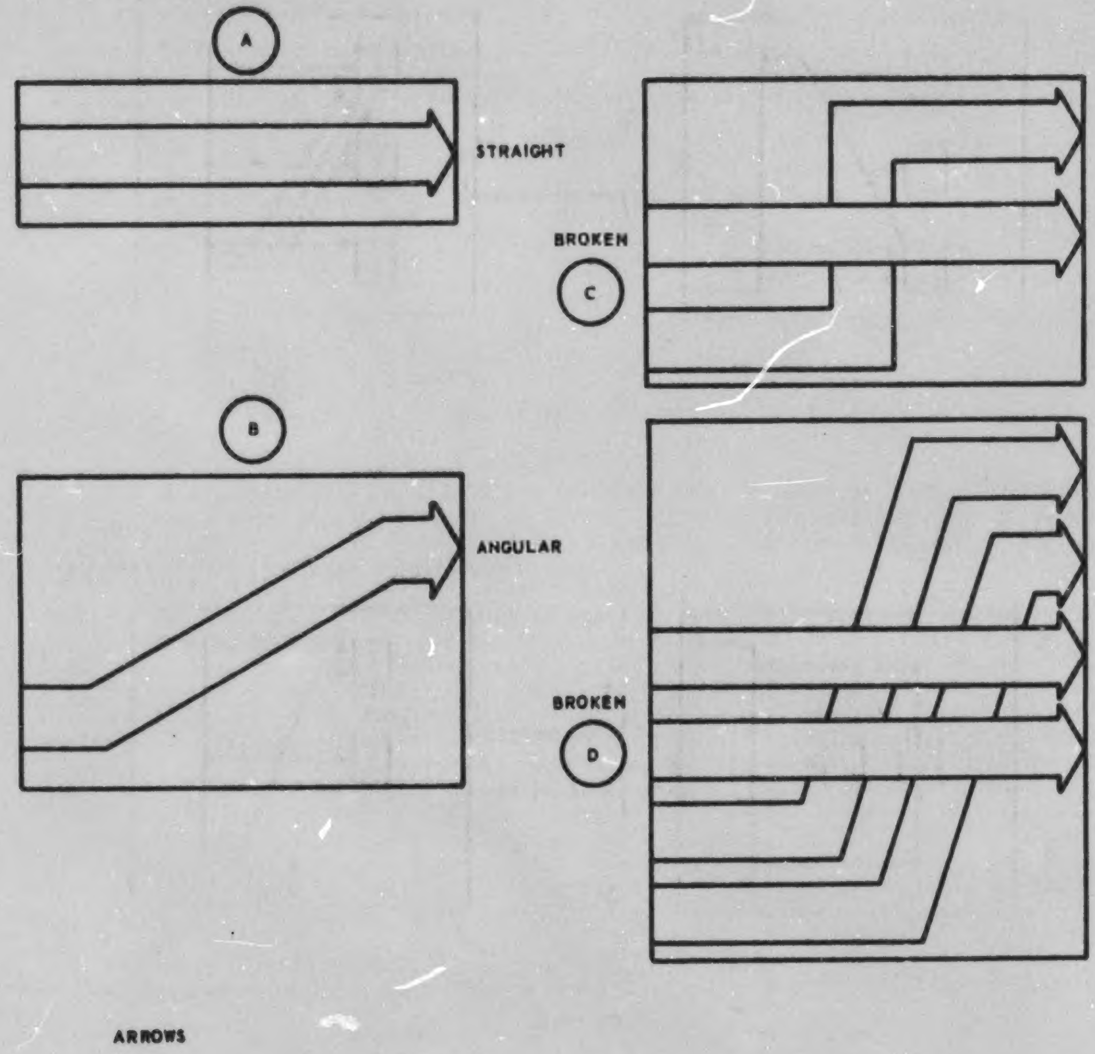
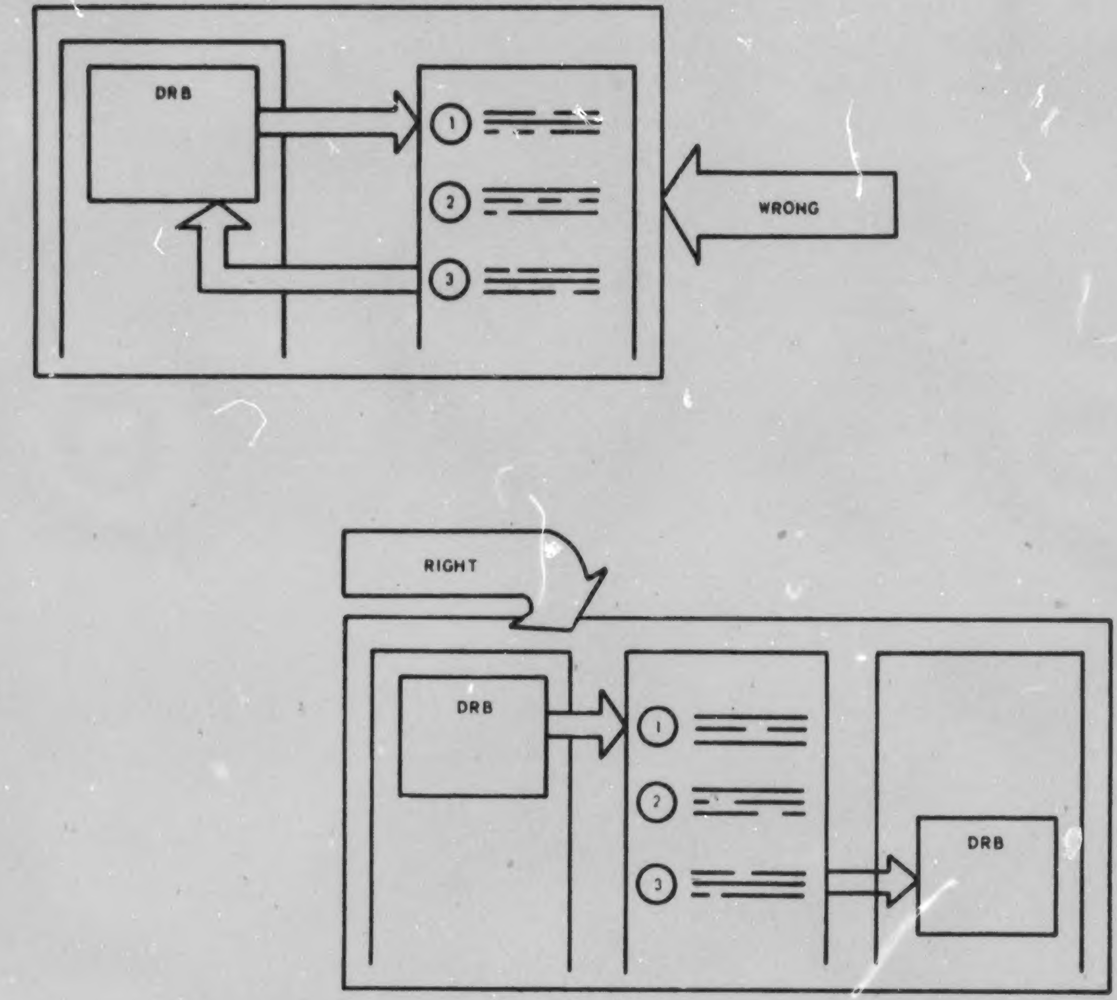


Figure 6-8

When an input is modified then it should be shown as an output (see figure 6-9).



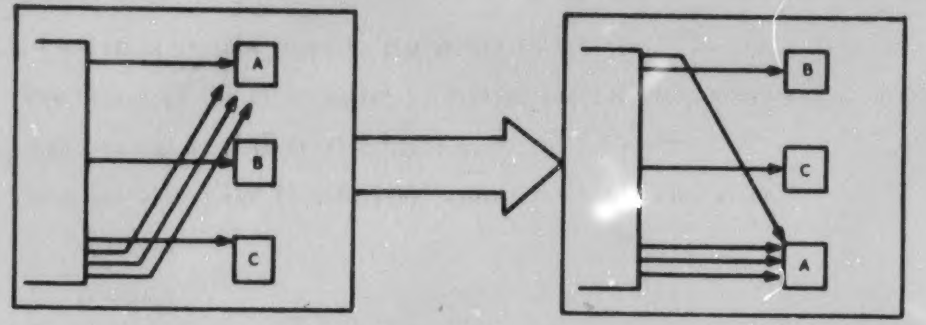
HOW TO SHOW THAT
A BLOCK IS UPDATED

Figure 6-9

Sometimes it is useful to do some rearranging of the diagram, known as tuning, to provide additional clarity. Figure 6-10 illustrates this tuning process.

1. COARSE TUNING

MOVE INPUTS AND OUTPUTS TO ALIGN THEM WITH THE AREA OF MAJOR ACTIVITY:



2. FINE TUNING

ELIMINATE CROSSED LINES BY MOVING INPUTS OR OUTPUTS TO ALIGN THEM HORIZONTALLY

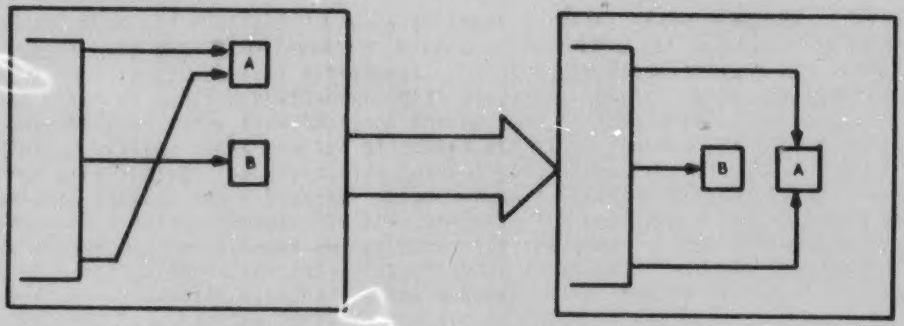


Figure 6-10

HIPO and System Development

HIPOs should be used throughout the development of the system. The initial design package provides an overall functional design of the system and is used by system designers to express their ideas. The initial design package is used by management in the preliminary reviews and by the designer/analyst to refine the initial design into a detail system design. The detail design package is used and supplemented by the analysts and programmers. During system implementation, the detail design package is used as a formal guide for coding. As sections are coded, the extended description may be enhanced with program labels and other important information to provide completeness. At the completion of the system implementation, the project has a complete documentation package. The detail design package can be used during system maintenance, but extensive system modification will require the use of a maintenance package. The maintenance package normally consists of the information in the detail design package minus the low level details used during implementation. Because HIPOs provide valuable system documentation, it is important to keep them current through the life of the system.

EXERCISES

1. Define HIPO.

2. What three diagrams make up the HIPO package?

3. Explain the purpose of each diagram in the HIPO package.

4. Explain the purpose of graphics in a HIPO package.

5. At what point in system development should HIPOs be used?

6. How long should the HIPOs used in system development be retained?

CHAPTER 7

STRUCTURED FLOWCHARTS

Upon completion of this chapter, you should be able to:

1. Name three of the five control primitives required for structured programming.
2. Name the two methods of flowcharting.
3. Draw the structured flowcharting symbol which represents:
    a. Sequence
    b. DO-WHILE
    c. IF-THEN-ELSE

STRUCTURED FLOWCHARTS

For years, flowcharts have been the primary method of describing procedural logic within a subprogram, module, program, or system. Flowcharting will continue to be used by some organizations although emphasis is now on structured flowcharts which use only well-defined control logic primitives.

Corrado Böhm and Giuseppe Jacopini in their article, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," proved that all programs can be written using only three basic control primitives. Using these primitives, a new method of flowcharting, known as Nassi-Schneiderman (N-S) diagrams, Chapin charts, or structured flowcharts was designed. The flowcharting method used is immaterial, but what is important is structured design. This chapter discusses the basic control primitives needed for structured programming. In explaining control primitives, we will use the traditional flowcharting symbols and a Program Design Language (PDL). PDL will be discussed more extensively in a later chapter. In this chapter, PDL provides a common basis for understanding and should not present any problems for the reader. Our discussion on structured flowcharting shows the Nassi-Schneiderman diagrams, named for the two men who were the first to publish material on the subject. The chapter is completed by comparing this new flowcharting method with the traditional method.

The basic control primitives, as described by Böhm and Jacopini, consisted of sequential processing, a general looping mechanism, and a two-state decision mechanism.

Sequential processing is the execution of the next logical instruction in the program as shown in figure 7-1.

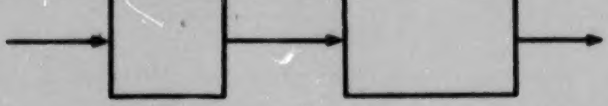


Figure 7-1

Iterative execution allows for repetitively executing a sequence of one or more instructions as shown in figure 7-2. It is normally stated in PDL as the DO-WHILE with the test for loop termination first.

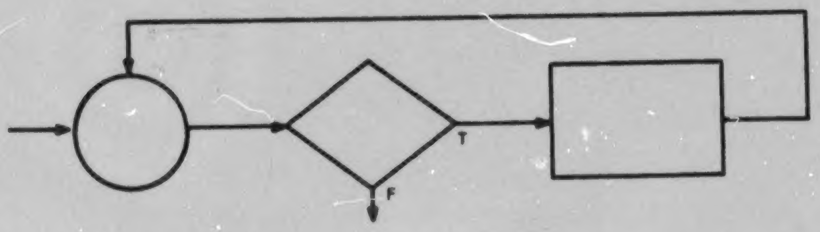


Figure 7-2

The two-state decision mechanism, or conditional execution, is typically known as the IF-THEN or IF-THEN-ELSE (IFTHENELSE) primitive. Figure 7-3 shows this primitive graphically.

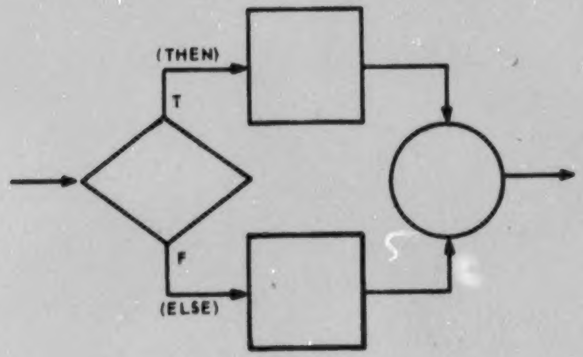


Figure 7-3

While it has been shown that these three primitives (also called constructs) are the only ones required for logical control of a program, two other constructs provide some additional flexibility in structured programming.

The DO-UNTIL construct is another looping mechanism which allows for the loop termination to be last as shown in figure 7-4.

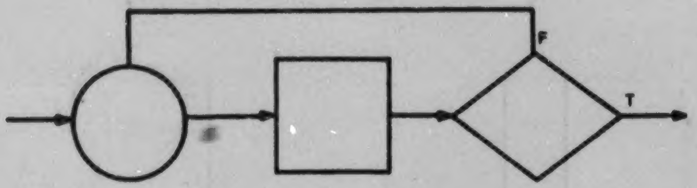


Figure 7-4

Program design sometimes requires a decision to be made from many possible states. In PDL, this construct is known as the CASE statement. It allows for the execution of one condition from a group of possible alternatives. Conceptually, the CASE statement is a series of IF-THEN-ELSE statements. Figure 7-5 expresses the case statement construct graphically.

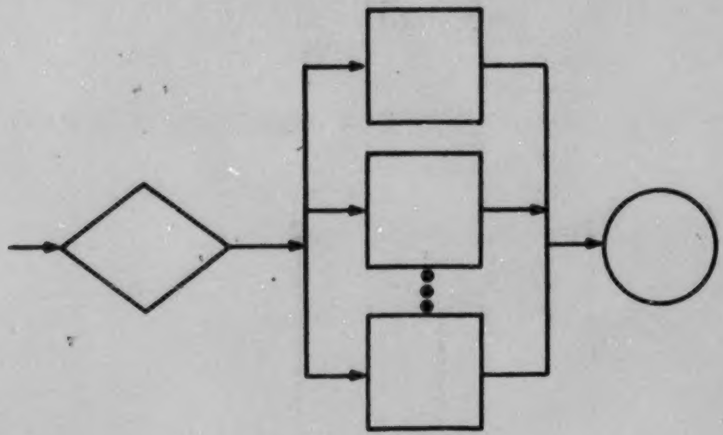


Figure 7-5

All of these well-formed primitives have been described graphically with traditional flowcharting symbols. While traditional flowcharting is one method of graphically representing a logical process, the Nassi-Schneiderman diagram (also called Chapin chart or structured flowchart) is another. The basic symbol in an N-S diagram is the box, which can be opened or closed. (See figure 7-6.)

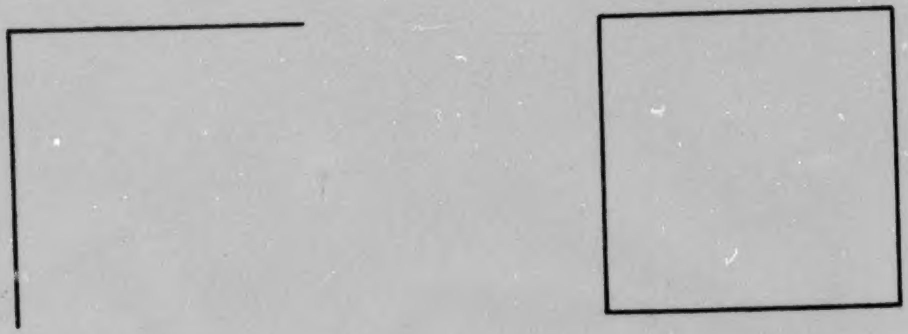


Figure 7-6

As the program is constructed, the box is dissected into the basic control primitives.

The sequential execution primitive is a horizontal line in the box with the verbal explanation of the process (see figure 7-7).

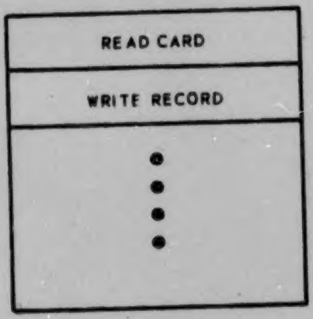


Figure 7-7

The looping mechanism is shown pictorially in figure 7-8. The test normally appears at the top with the instructions below the test (figure 7-8a). It is possible to have the test following the instructions as shown in figure 7-8b.

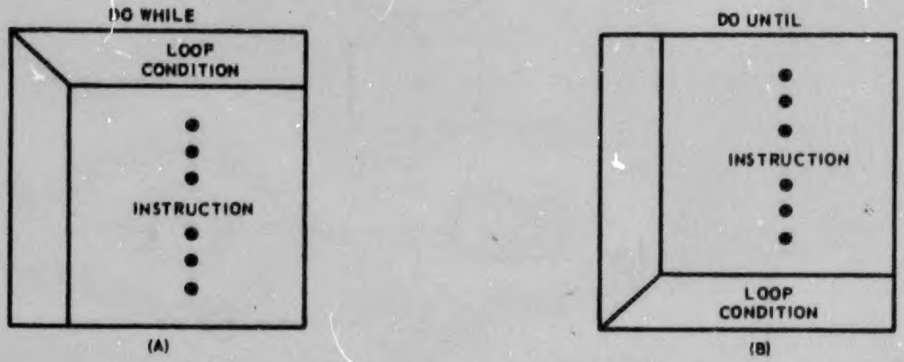


Figure 7-8

The decision, or IF-THEN-ELSE, which is difficult to explain verbally, is shown in figure 7-9.

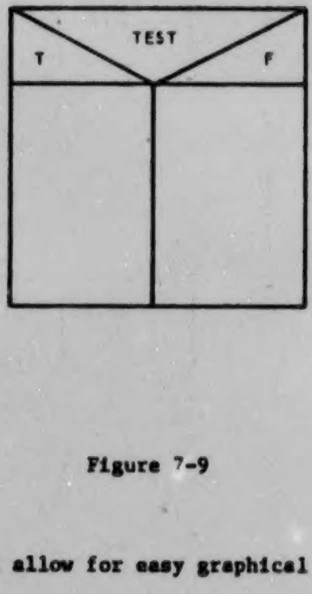


Figure 7-9

The CASE statement does not allow for easy graphical representation and has not been formalized.

Logical construction of a program often requires one basic primitive to be used in conjunction with another (commonly referred to as nesting). The following example illustrates how nested control structures are handled in the N-S diagrams (figure 7-10).

```
IF        (cond p) then
       Do While (cond r)
          ______
          ______
            :
       END DO
ELSE
  IF      (cond s) then
       DO UNTIL (cond t)
          ______
          ______
            :
       END DO
  ELSE
          ______
          ______
  END IF
END IF
```

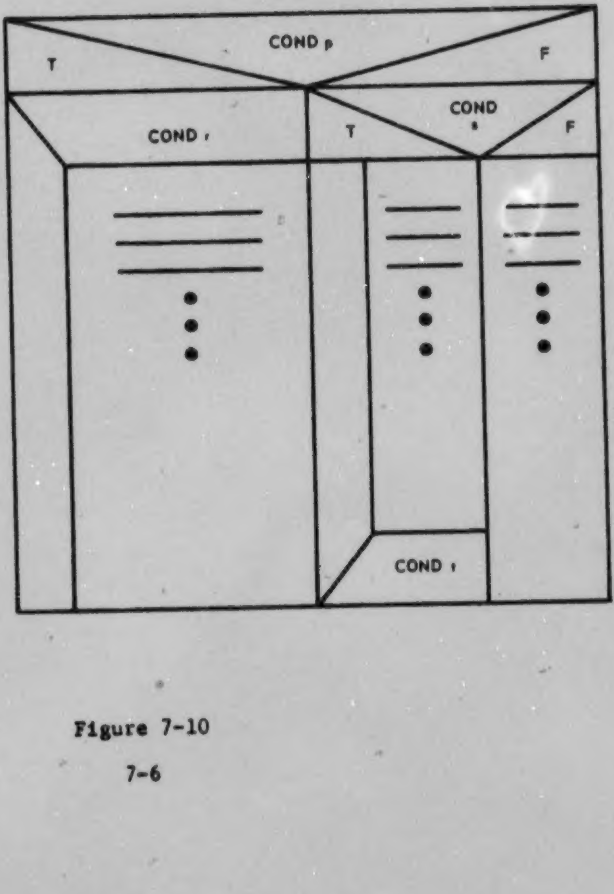
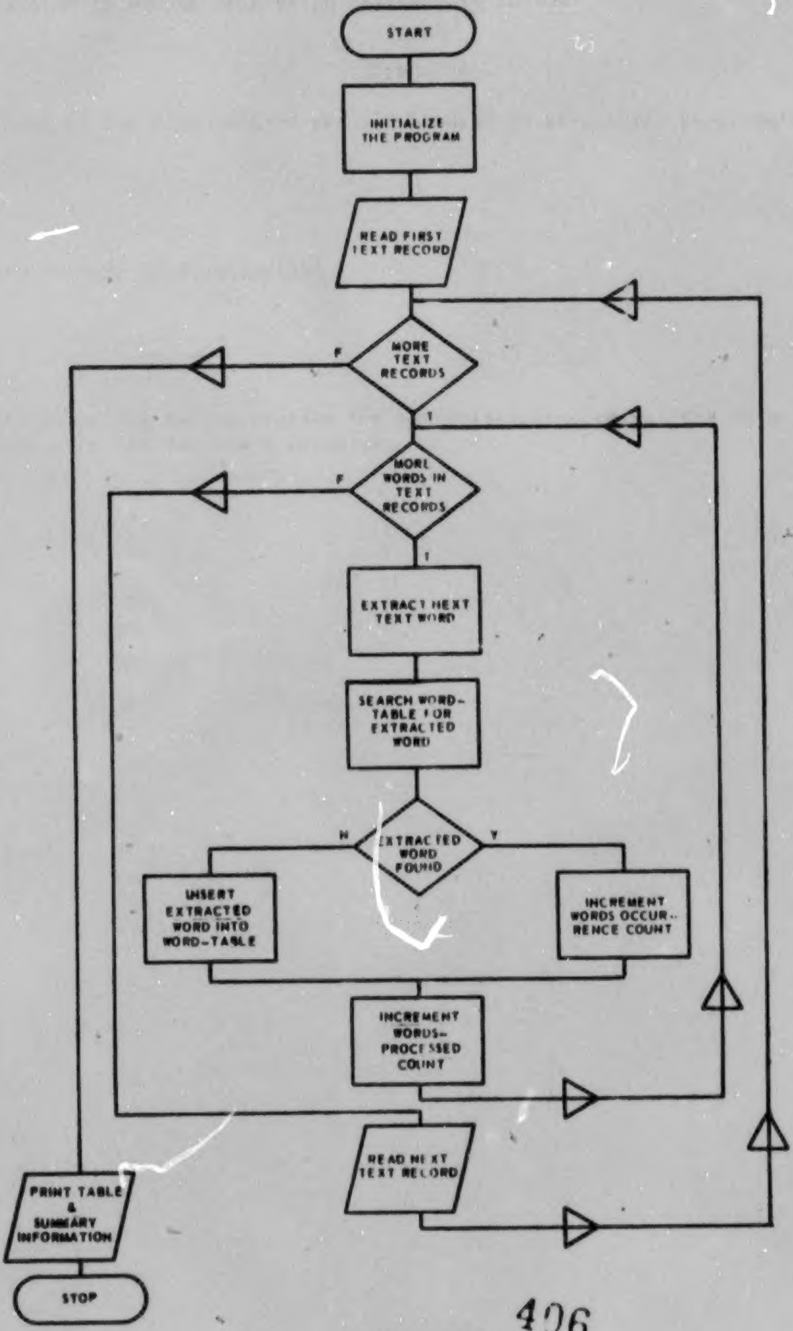


Figure 7-10

The word frequency analysis program is shown by both methods of flowcharting. Both methods are well-formed and it should be programmer discretion as to which to use. Figure 7-11 illustrates the conventional flowcharting method while figure 7-12 shows the same logic in a structured flowchart (N-S diagram).


START
INITIALIZE THE PROGRAM
READ FIRST TEXT RECORD
MORE TEXT RECORDS
F
T
MORE WORDS IN TEXT RECORDS
F
T
EXTRACT NEXT TEXT WORD
SEARCH WORD-TABLE FOR EXTRACTED WORD
EXTRACTED WORD FOUND
N
Y
INSERT EXTRACTED WORD INTO WORD-TABLE
INCREMENT WORDS OCCURRENCE COUNT
INCREMENT WORDS-PROCESSED COUNT
READ NEXT TEXT RECORD
PRINT TABLE & SUMMARY INFORMATION
STOP

406

Figure 7-11

WORD FREQUENCY ANALYSIS PROGRAM

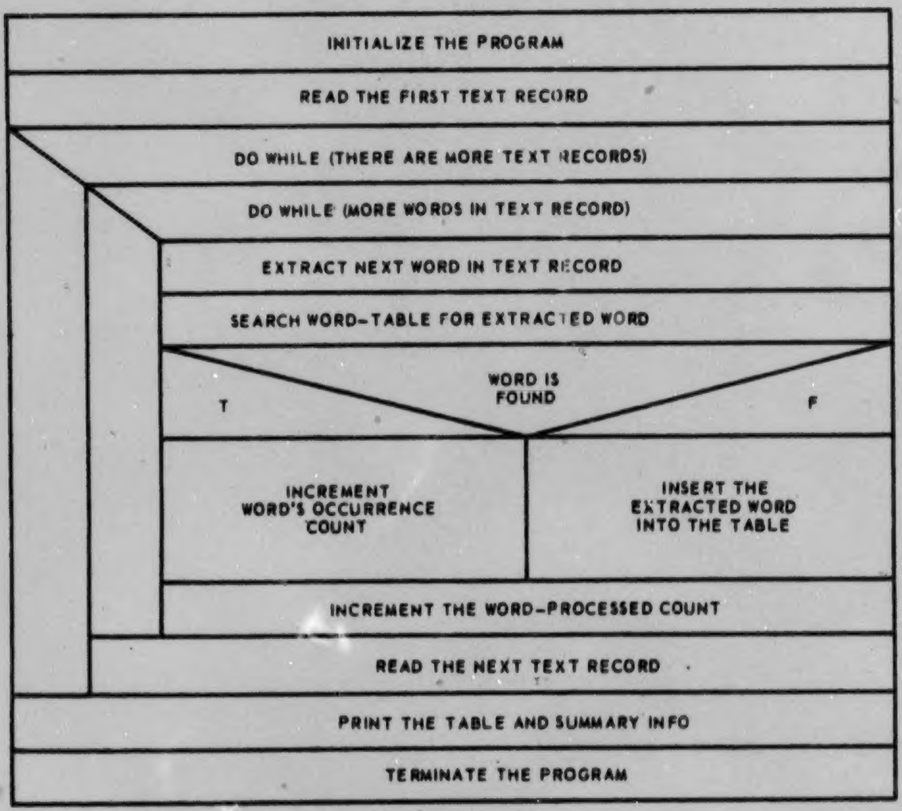

INITIALIZE THE PROGRAM
READ THE FIRST TEXT RECORD
DO WHILE (THERE ARE MORE TEXT RECORDS)
DO WHILE (MORE WORDS IN TEXT RECORD)
EXTRACT NEXT WORD IN TEXT RECORD
SEARCH WORD-TABLE FOR EXTRACTED WORD
WORD IS FOUND
T
F
INCREMENT WORD'S OCCURRENCE COUNT
INSERT THE EXTRACTED WORD INTO THE TABLE
INCREMENT THE WORD-PROCESSED COUNT
READ THE NEXT TEXT RECORD
PRINT THE TABLE AND SUMMARY INFO
TERMINATE THE PROGRAM

Figure 7-12

414

The structured flowcharting method appeals to many people because of its simple approach to logical program development. An opposing viewpoint to N-S diagrams is that it resembles a program design language enclosed in boxes, so why not write the program design language and forget the diagrams. Both viewpoints are valid and it remains up to each organization to decide upon which approach to follow.

EXERCISES

1. List three of the five control primitives used in structured programming.

2. List two methods of flowcharting.

3. Draw the graphical representation for sequential processing, the DO-WHILE looping mechanism, and the two-state decision.

408

Chapter 8

DESIGN TECHNIQUES

INTRODUCTION

The previous chapters in the design section explored the various graphical representations for data flow graphs, structure charts, HIPOs, and Nassi-Schneiderman diagrams. Becoming familiar with the mechanics of these tools provides for some standardization, thus eliminating interpretation problems within an organization. While it is important to have consistent interpretation by organization members, the tools, themselves, provide little toward a good design. A good design is the result of applying a good design strategy. There is a brother-sister relationship between the various tools and strategies, in that, the tools help formulate the design strategies visually. There are many schools of thought in existence, each of which provide a different avenue of attacking the problem. The greatest benefit comes when they are taken together to represent a complete design solution.

This chapter discusses some strategies which aid in the design process. Transform analysis studies the data flow through the system and is represented via the data flow graph discussed in Chapter 4. Coupling and cohesion address the relationship between modules and within modules, respectively. Structure charts are used to decompose a problem into smaller and simpler processes and, as a result, the coupling and cohesion analysis is usually performed from the structure charts. Finally, transaction analysis, which attacks the design from a decisions up - detail down approach, can utilize either HIPOs or structure charts.

Upon completion of this chapter, the student will be able to:

1. Define module coupling.
2. Explain what influences module coupling.
3. Explain the difference between local and global parameters.
4. Apply the complexity formula and interpret the results.
5. Define normal and pathological connections.
6. Explain the best type of module communication.
7. Define module cohesion.
8. Explain the six levels of module cohesion.
9. State the philosophy behind transaction analysis.
10. Draw the basic structure charts for transaction analysis.
11. Define transform analysis.
12. Explain the general organizational structures produced by transform analysis.
13. List the four steps of transform analysis.

14. Apply transform analysis to simple problems.

15. Define factoring.

16. List the four guidelines for organizing solutions.

TRANSFORM ANALYSIS

Transform analysis is a strategy for designing highly modular programs and systems by studying the data flow through the problem. The basic idea of transform analysis is to design a program so that a central module calls for highly processed logical data. Eventually it will generate logical output data. Other related modules transform physical input to logical input and logical output to physical output.

This generally leads to a solution organized as in figure 8-1 where the INPUT module is in charge of transforming the physical input data into logical input data (X). The PROCESS module has the responsibility of transforming the logical input (X) into logical output (Y). The duty of the OUTPUT module is to take the logical output (Y) and produce the required physical output.

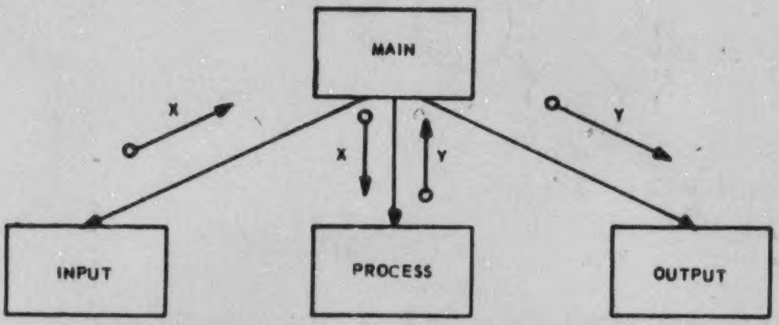


Figure 8-1

Problems which can be described as inputting data, processing it, and outputting the results consistently generate a solution organized as in figure 8-1. Regardless of the type of problem, transform analysis will provide some clues and steps to follow for developing an effective yet flexible solution organization.

Transform analysis starts with a data flow graph of the problem. The data flow graph should be detailed enough to show the major parts of the problem, but not so detailed that the graph becomes cluttered and difficult to use. About 5 to 15 bubbles is a ballpark figure, although there are problems which have been satisfactorily illustrated outside of this range. The figures are offered only as a guide to the novice at transform analysis.

Remember that a data flow graph is an interpretation of the problem. A general understanding of the problem is needed before details are added. Experience will dictate the meaning of "sufficient detail" in a data flow graph.

The second step in transform analysis is to identify the streams of input and output data. An input stream is identified by following the input data into the data flow graph until the data item is no longer viewed as input. The output stream is identified by starting at the end of the data flow graph and following the data backwards to the point where the resulting data are no longer viewed as output. The last data item on an input "leg" which is the most abstract item still associated with the input, marks the end of an input stream. The first data item to be viewed as output in its most abstract form is the start of the output stream.

For example, figure 8-2 is a bubble chart for a particular problem which has an input requiring various transformations to obtain the required output. Analyze the data flow graph and mark the input stream and the output stream.

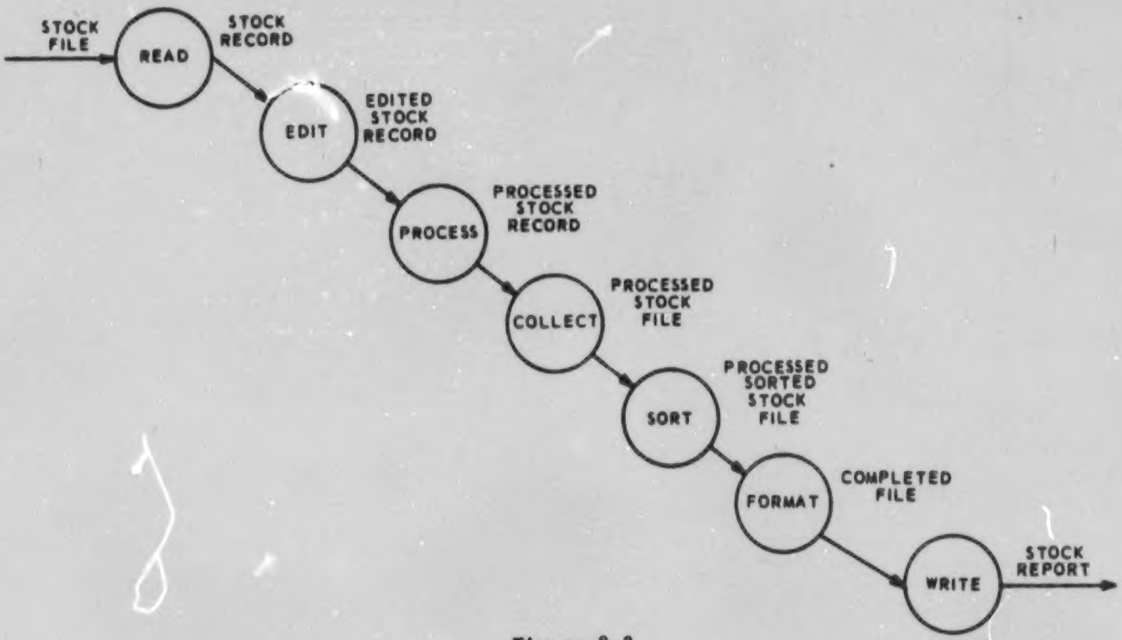


Figure 8-2

The general opinion is that the "edited stock record" is the last data item which can be viewed as input. This, then, would mark the end of the input stream. The consensus is that "processed stock file" is the first data item which is viewed as output for the overall problem, so this would mark the start of the output stream.

The preceding paragraph presents some debatable opinions on a subjective exercise. This example is presented as a guide. Since people (even programmers) think differently, there will be differing opinions on the identification of input and output streams. The transform analysis strategy is sufficiently simple and flexible to allow different ideas to be evaluated.

418

Once the divisions have been made identifying the input stream(s) and output stream(s), the remaining bubbles are the central transform. (See figure 8-3.) This is where the abstract input data is transformed into the abstract output data.

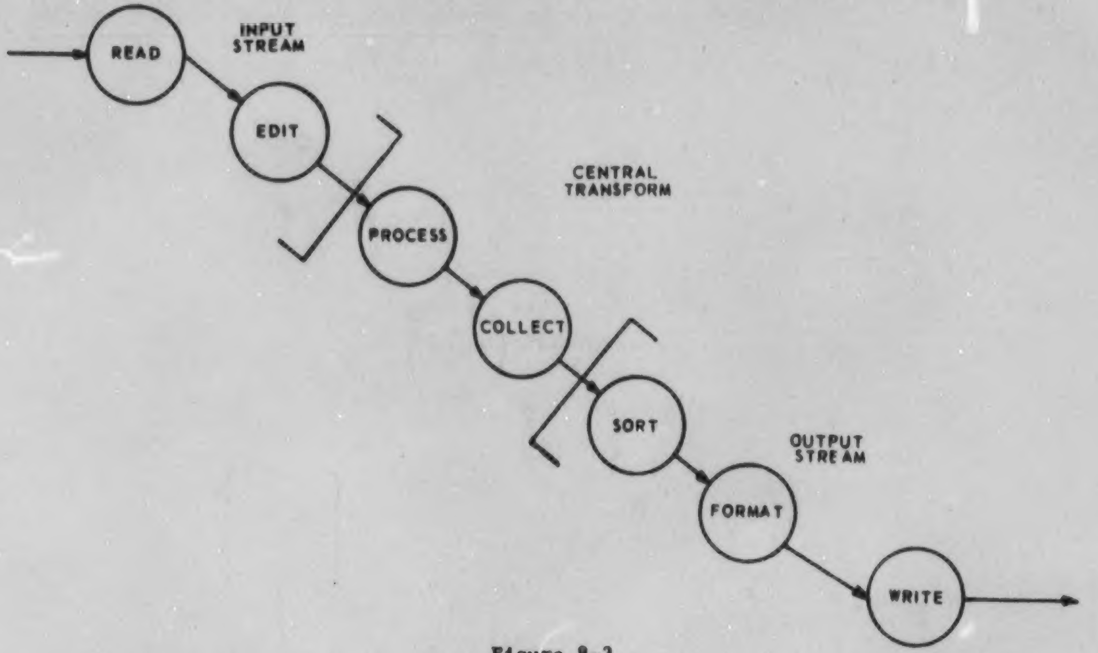
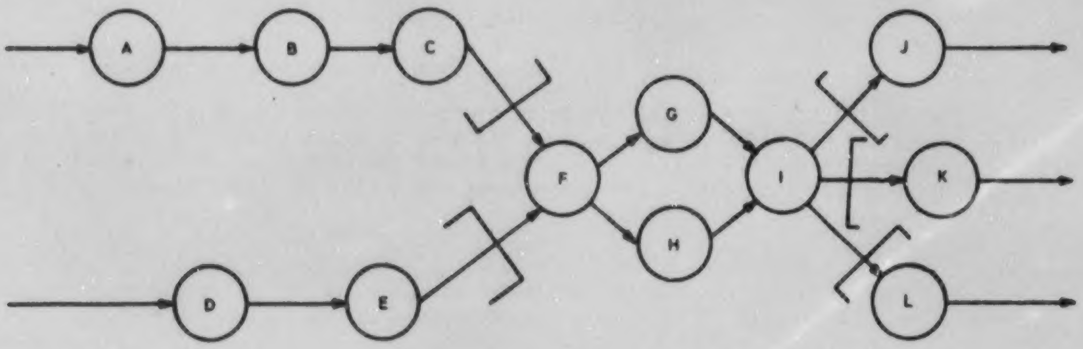


Figure 8-3

At this point, there are three main divisions in the problem: input, transform, and output (similar to HIPOs). Multiple input streams and/or output streams may exist as in figure 8-4. These can be organized by maintaining the three major divisions, input, output, and transform, or by having multiple input and output divisions in the organization.



412

Figure 8-4

Constructing the solution organization is the third step in transform analysis. This is normally accomplished with a structure chart, but more recent applications show that this is effectively accomplished with HIPOs as well.

To start the design organization, there will be one module responsible for the overall operation of the solution. In matching the design to the problem, this main module is THE solution at the highest level of detail. The function of this module is the same as the description of the problem at the highest level. Terms to describe this module are simple and encompassing such as EMPLOYEE PAYROLL or FILE UPDATE. If a data flow graph of the problem were constructed at such a high level that there was only one bubble, then the function of the bubble would be the same as the functional description of the main module as in figure 8-5.

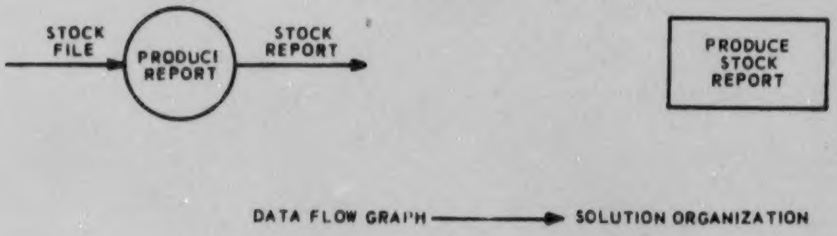


Figure 8-5

This main module will exist as the boss for all subsequent, more detailed, designs. Perhaps a 1-bubble data flow graph is a good way to establish and reaffirm the overall objective of the problem. This can provide a starting point for more detailed analysis. Before going on a trip, it is good to know the starting point and destination. Programmers sometime become detail oriented and lose sight of the objective, and either solve the wrong problem or reach a programming impasse.

Unfortunately, most programs do not end with the main module; that is, the problem is too complex to solve with one module that is small enough to be manageable. Then, the next level of design is required to separate the major functions necessary to solve the overall problem. This means that the main module will be directing a few subordinate modules to carry out the required work. This separation of functions is called functional decomposition, or more simply factoring. Factoring is now less an art since the emphasis on design tools and techniques, but there are still usually several good designs from the same problem, each with their own merits. Transform analysis is a procedural method to use to obtain a "good" design. The organizational process is based on the analysis of the data flow graph. For starters, each input stream and each output stream has a corresponding module which completely handles the associated data. There is also a module in charge of that central transformation. Then, the resulting structure at the second level of detail has the number of modules equal to the input streams, plus the output streams, plus one (the central transform). Figure 8-6 shows the solution to a problem with one input stream and one output stream.

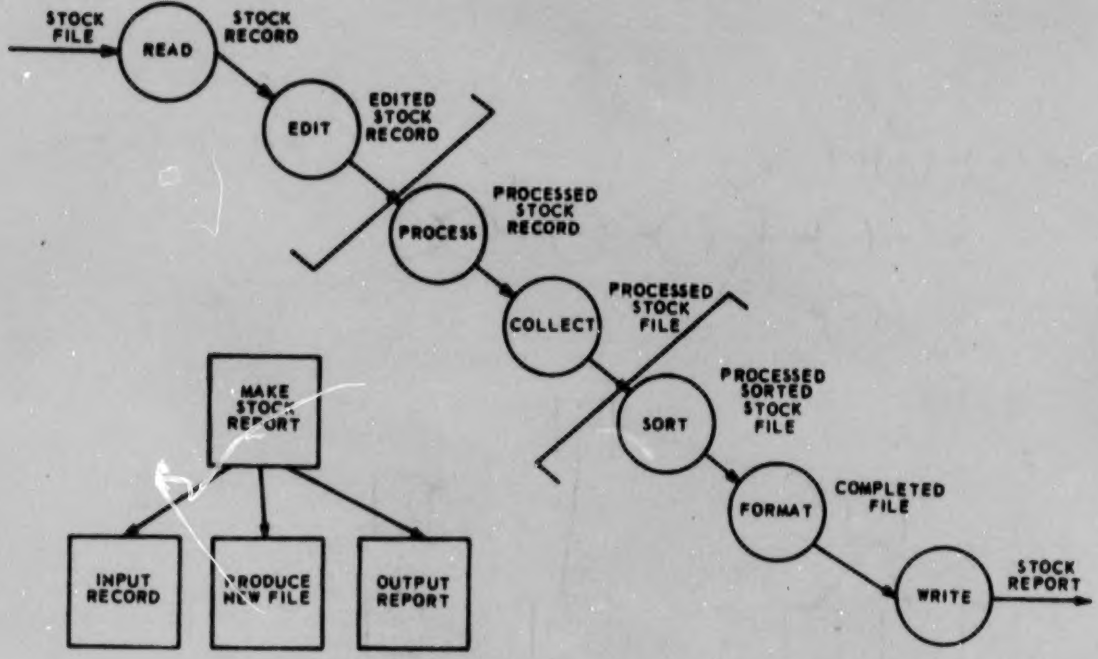


Figure 8-6

At each step in the design process, time should be spent in reflection, looking back to see if the organization is complete at each level. Only then can the final step in the transform analysis strategy take place. This is the process of repeatedly factoring modules until they are fully decomposed into single indivisible functions.

Each subordinate module in figure 8-6 can be further factored into its subordinate functions. This next level of factoring can be visualized as a transform analysis of the INPUT RECORD, PRODUCE NEW FILE, and OUTPUT REPORT modules. Indeed, each of these can be thought of as individual problems requiring a solution. If the data flow graph in figure 8-6 is correct and sufficiently detailed, then the continuation of the organizational solution would look like figure 8-7 at the third level.

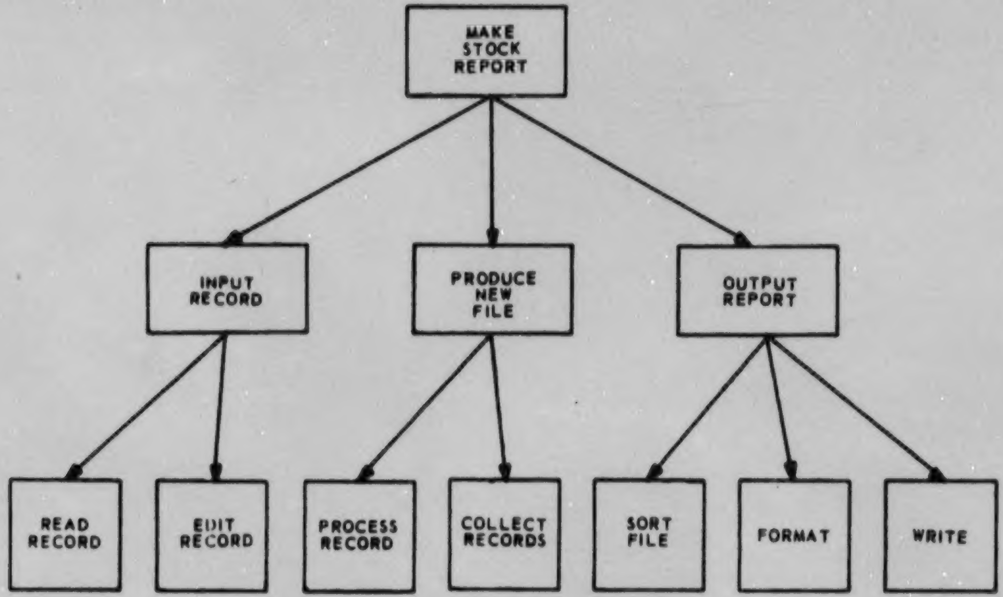


Figure 8-7

Problems which have multiple input and output streams can be organized with an equal number of managing modules as input and output streams. Another way of organizing a solution is shown in figure 8-8 where two new manager modules have been introduced to alleviate some of the work from the main module. This is a good way to reduce the workload and complexity of the main module. If, at a later date, it is decided that these modules are really not needed, they can be compressed into the main module. Notice that the position will be eliminated but their work must be done elsewhere.

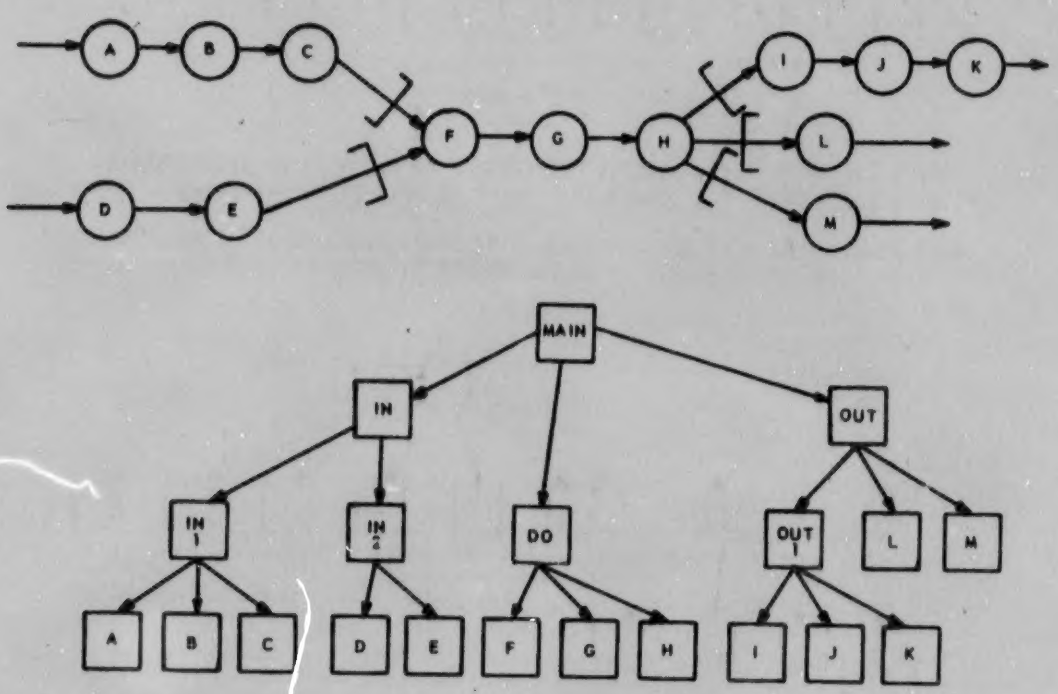


Figure 8-8

This is the concept of factoring, taking each module and dividing the functions into smaller, easier to handle problems. The problem of overfactoring is a rare exception. Usually the functions are not sufficiently detailed. It is easier to combine modules later in the development process, if modules are too simple, than to decide that a module is more complex than originally thought and then decide to factor further.

Simply stated then, the steps in the transform analysis strategy are:

1. Construct a data flow graph of the problem.

2. Identify the input and output streams, and the transform center.

3. Construct the corresponding organizational structure.

4. Repeatedly review the organization while subdividing the modules to produce a fully factored design.

The end result of the transform analysis is an organization which can coordinate the many activities involved in the problem solution. This organization is similar to management organizations where some management practices provide these useful guidelines for managing modules:

1. A manager should not have more than 7 ± 2 immediate subordinates.

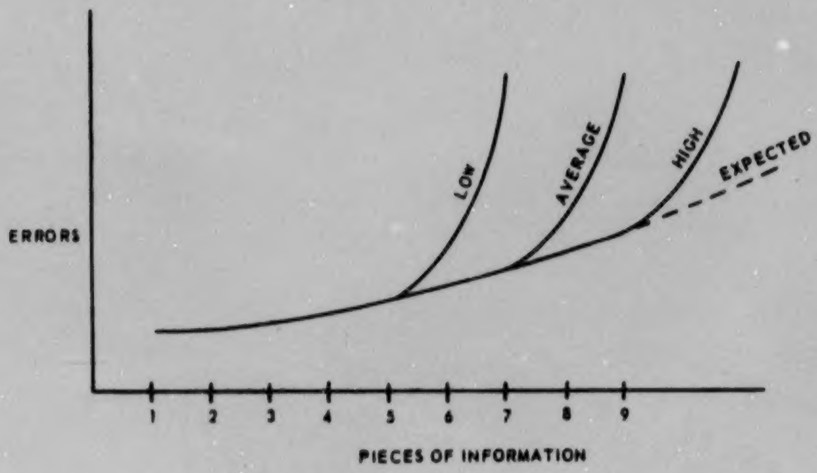


Figure 8-9

The average person can handle problems involving various numbers of pieces of information and naturally, the more complex the problem, the greater the number of errors, as expected. However, as illustrated in figure 8-9, the errors skyrocket when involving more than seven pieces of information. For some people, this upper limit on complexity is as low as five, for others as high as nine.

424

What then should be done with solutions organized as in figure 8-10?

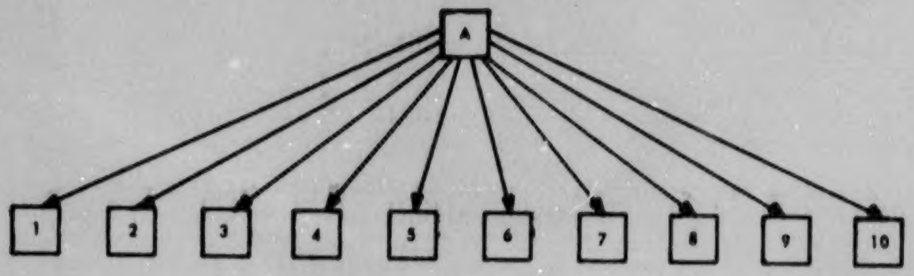


Figure 8-10

The easiest thing to do would be - nothing. Indeed, this may be the correct action. Module A may be simple and efficient as it is. Remember, 7 ± 2 is only a guideline.

Consider some other alternatives. If several modules have some common bond, then there may be one or more submanagers introduced as in figure 8-11.

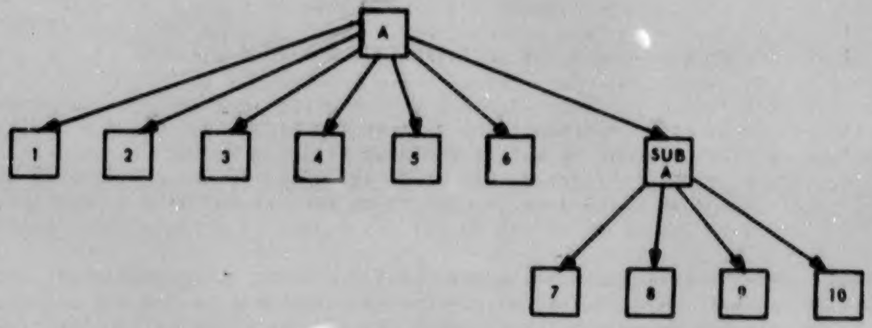


Figure 8-11

In most problems involving a large number of pieces of information, there are usually natural groupings creating submanager positions which prevent this complexity problem.

2. Maintain communication in the chain of command. A module should "communicate" only with its immediate superordinate(s) and its immediate subordinates. This adds clarity to the solution and prevents the harmful side effects of changing a module and

418

discovering two weeks later that the change also had some hidden far-reaching effects on other parts of the program. A violation of this principle is the pathological connection discussed in Chapter 5.

3. Managers should make decisions which affect only their subordinates. Similarly, modules which are affected by a decision should be subordinate to the module which makes the decision. This design consideration has a tendency to move decisions up in the organization to resolve this "effect vs control" conflict. This also results in the upper level modules making decisions while lower level modules do the work. Good! After all, this is the way an organization should operate.

4. This management parallel is the somewhat controversial "mushroom theory" where a subordinate is only given the information necessary to do his task.

This concept is applicable to design organization since it is important to maintain complete control of information or data flow throughout the problem solution. Violators of this principle run the risk of mysterious appearance or disappearance of data, causing unexpected results. Programmers that say they have not experienced this situation probably lie about other things also.

The transform analysis strategy is stressed as a method for obtaining a functional design which reflects the problem. If the problem changes (there is always that remote possibility), then the solution will change accordingly, with a minimum effect on the parts of the solution which are not altered. This means that a change has less chance of causing a major rewrite.

The organizational design should be developed with the thought of change in mind. One of the major items of concern should be to strive for modules which are highly functional; that is, they do a specific task the same way every time. Another desirable quality is for modules to be as independent as possible. The module which can "stand alone" will probably not be affected by changes in other modules.

The concepts of modules being independent and functional will be discussed in detail to explain how to arrive at the most cost-effective system possible.

COUPLING

The relationship between modules and within modules provides valuable information for the designer. This information is used to determine how complex the system is becoming. If the designer makes a sincere effort to keep the system as simple as possible, then a change to the system will require less effort to accomplish. W. P. Stevens, G. J. Meyers, and L. L. Constantine are credited with much of the research in this area. They termed intermodule relationships "coupling" and intramodule relationships "cohesion." Any system designing process should strive for as much independence between modules as possible and modules that are as functional as possible. What is desired then are modules that perform a specific function and rely very little on other modules in the systems, thus producing Reliable Modifiable Software.

Coupling is defined as the measure of strength of association between one module and another. Module interfacing, type of module connection, and type of communication need to be studied to determine module coupling.

Module interface is concerned with how modules retrieve their data for processing. Modules receive data basically by two means, local and global parameters. Local parameters are passed directly to the module that needs them from the supplying module and thus are localized within the two modules. It is desired that parameters be passed by the parameter passing mechanism which best localizes the data. An example of this is the Fortran call statement using the parameter passing capability. (See figure 8-12.)

Call Foo (A, B, C)

Figure 8-12

Global parameters are normally supplied at a system level. The access restrictions can vary from complete availability to very limited availability. Cobol's data division is a global parameter passing scheme where variables are available to all subprocesses in the system. The use of the Fortran common statement is another way of providing global parameters. By using the named common convention in Fortran, the programmer can reduce the availability of data to a small subset of the system. While local variables are considered loosely coupled (which is good), global parameters are considered highly coupled (not so good). The primary reason is that as data elements become more available to modules in the system, the interface between those modules becomes more complex. When a data item is changed, all modules using the data item must be considered in determining the effects of change. The interface complexity can be summarized by the formula:

$$C = E \times M \times (M - 1)$$

C = Complexity

E = the # of elements in the common region

M = the # of modules utilizing the common region

For example, suppose that two modules, A and B, access a common data region with two variables, X and Y, the total number of paths (complexity) is four. Module A can affect X and Y as can module B. This does not appear to be very complex, but suppose the common data region has 25 variables in it and is shared by three modules. It is important to make module interfacing as obvious as possible. As module interfaces become obscured, the coupling is higher and system change is harder to make.

Normal and pathological connections are the major module connections. Normal connections are defined as references to modules by name with no direct reference to that module internally; that is, internal elements and processes are irrelevant to the normal connection. If module A calls module B to perform a function, whether or not module A passes parameters to module B, the connection between them is considered normal. Normal connections tend to yield lower module coupling. Even though normal connections are considered the best, as the number of parameters increases so does the coupling. It is important, then, to pass only those parameters needed for the performance of the task. Do not degrade an otherwise good connection with extraneous information.

Pathological connections make internal references to a module. Simply stated, this means that module A makes reference to a program element (either data or control) in module B. Pathological connections produce highly coupled modules because any change made to one module may affect the pathologically connected module which will also need to be changed. Pathological connections should be avoided if possible.

Module communication is concerned with the type of data being passed. If a module does not pass or receive data within a system, then it is not a functional part of the system. The best type of parameter to pass is strictly a data parameter. Down the line comes control variables and at the bottom of the spectrum is hybrid variables. Data coupled modules are extremely good modules due to the lack of control of one module by the other. Module A calls module B, which performs a function and returns. This is a very simple and obvious type of coupling which is easy to visualize. Both modules are very independent; that is, a change to either module will have little likelihood of affecting the other. Control parameters, such as flags and switches, result in higher coupling. Any time one module controls the execution of another, the coupling is increased. Why? Because a change to the calling module has a greater chance of impacting the called module. It is necessary to realize that control coupling adds another degree of complexity to the system. It is possible, in most systems, to avoid control coupling by rethinking the design of the problem. Hybrid coupling occurs two different ways. It occurs when program instructions are changed during execution (i.e., the Cobol alter statement). Changing program instructions during program execution produces a very volatile system which is hard to understand, debug, and maintain. The other type of hybrid coupling occurs when a parameter is used for both data and control. While this is not obvious, it happens quite frequently in program design. A hybrid parameter, when passed to a module, is used for decisionmaking and also for calculating a result. The danger of this type of coupling should be apparent and therefore avoided.

COHESION

Another valuable relationship to consider, when designing a system, is the internal strength of a module. L. L. Constantine has termed this characteristic as cohesion. It is also known as module strength and module binding.

The designer should strive for functional modules which perform only one task. Modules tend to be more complex as more and more tasks are performed in them. There are varying degrees of cohesion ranging from weak module strength to strong module strength. If the cohesiveness of modules is low, there is a good chance that major problems will arise throughout the life of the system. This is not to say that problems won't occur in a functionally strong system, but the problems tend to be very minor. The following discussion is designed to give the reader a basic knowledge of the various levels of cohesion as an aid in deciding the strength of a module. They are not to be considered formal rules which clearly place a module into a specific category, rather, they should be used as guidelines to narrow the realm of possibilities. It is not always possible to precisely define the strength of a module, but being in the ballpark does provide valuable information as to the type of modules being designed, which ultimately indicates how well the system is being designed.

"Structured Design," an article written by W. P. Stevens, G. J. Meyers, and L. L. Constantine, defines the six levels of cohesiveness, in order from bad to good, as coincidental, logical, temporal, communication, sequential, and functional.

A coincidental module is just what the term implies. The module is a hodgepodge of instructions which have absolutely no relationship to each other. This type of module is considered very weak and extremely unstable and usually very easy to spot. It occurs primarily when trying to save memory or when making an artificial attempt at modulization. For example, suppose the following instructions appeared individually several times in a program (figure 8-13).

```
Read (G, J) A, B, C

I = I + 5

DO 1 J = 1,5

GLOP (K) = FLOP (K)
```

Figure 8-13

If grouped into a module to reduce the number of separate occurrences, then the module would be considered coincidental as there does not seem to be any apparent relationship among the instructions. Coincidental modules can and should be avoided in a structured programming environment.

Logically bound modules are one step removed from coincidental modules and imply a logical relationship among module elements. Suppose that module X edits all data and furthermore the types of edits include master edit, update edit, addition edit, and deletion edit. Including these edits in one module will require not only the data to be edited, but a flag to specify which edit to perform. Because the entire module deals with an editing function, it is considered logically cohesive. Normally, a logically cohesive module will execute only a portion of the module each time it is called. If at all possible, it is a better design to break a logical module into functional units.

Temporally bound modules have time-related elements. Typical modules in this category are initialization and termination routines. Modules that are temporally bound tend to be less complicated than logically bound modules because of the absence of control variables. When called, the entire module is usually executed.

Modules with communicational cohesion are characterized by referencing the same set of data. For example, "print and punch the master file" references the same output file for printing and punching and is, therefore, communicationally bound. Referencing the same data structure produces a strong bond between module elements.

Occasionally, the output from one module element is input to the next module element. The binding for such a module is termed sequential. A sequentially cohesive module may perform either some part of a function or several subfunctions. Because of this characteristic, a sequentially bound module is still far removed from the more desirable functional module. The utility of sequential modules in other parts of the system is usually somewhat low. However, the module, itself, is still considered cohesively strong.

The strongest type of module strength is the functional module. Modules which perform only one task are considered functional in nature. These modules are very stable, easy to maintain, and usually integrate into the system with very few problems. The major decision is how far to divide functionally bound modules. A good criterion to use is when each module contains no subset of elements that could be useful alone and is small enough that its implementation can be dealt with at one time.

A useful technique for determining the strength of modules is to describe the purpose of the module in a sentence.

1. If the sentence is a compound sentence, then the module is probably sequential or communicational.

2. If the sentence contains words relating to time, such as first, next, or after, the module could have sequential or temporal cohesion.

3. If the sentence is missing a specific object to describe the module, the module is probably logically bound.

4. Words such as initialize and cleanup imply temporal binding.

Module coupling and cohesion should be considered during system design. Even though the techniques have been discussed as an after system design process, the analyst should be aware of the type of modules being designed as the system is built and should strive for a functional design.

TRANSACTION ANALYSIS

Transaction analysis is a particularly useful design technique for various data processing applications. The strategy promotes a decisions up, work down philosophy. Just its name implies that it would be extremely useful in the business data processing community because most of those problems deal with some form of transaction. By extending the definition of a transaction, the technique provides design support in other data processing areas. In particular, a transaction is any type of program element which causes a specific sequence of actions to be performed. An element of data or control which flows into a process and then proceeds to one of a set of action sequences is a candidate for transaction analysis. This situation is illustrated with the use of a data flow graph in figure 8-14.

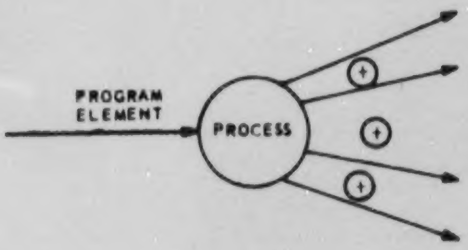


Figure 8-14

The PROCESS in figure 8-14 analyzes the PROGRAM ELEMENT to determine which action sequence path to follow (note the OR symbols used in the data flow graph).

Once it is determined that transaction analysis is to be used, the appropriate structure chart to use is shown in figure 8-15.

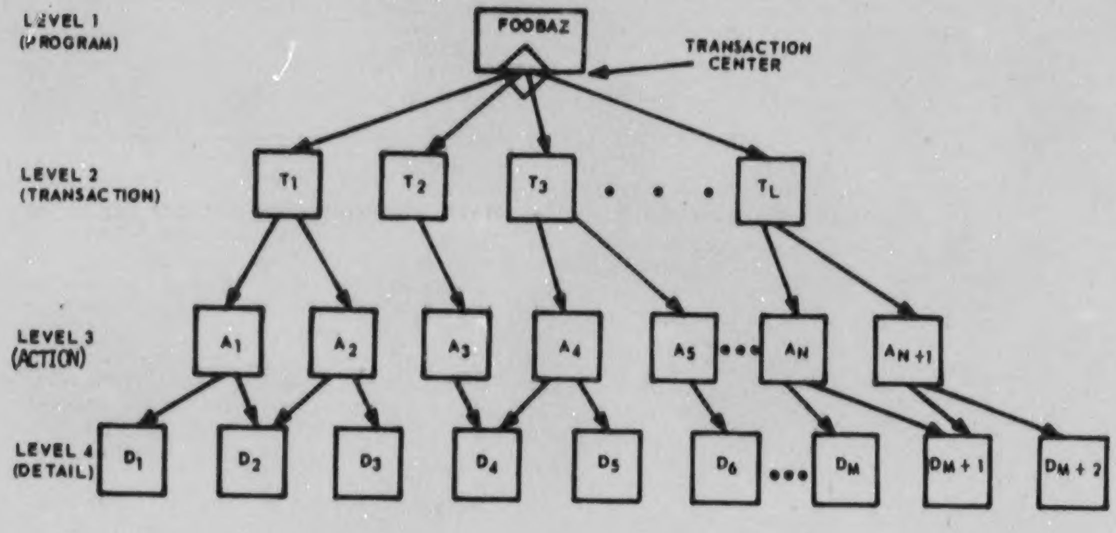


Figure 8-15

The acronym SAPTAD (System, Analysis, Program, Transaction, Action, Detail) expresses the original philosophy of Transaction Analysis. The first step is system analysis followed immediately by creation of the Program, Transaction, Action, and Detail. Each level in the chart represents an additional level of detail beginning with the program. The first level (program module) is the transaction or decision center. This module usually does the dispatching of transactions. The second level receives an appropriate transaction and processes the specific actions (third level) necessary for that transaction. Each action module uses detail modules which do most of the work.

The structure is derived from the original concept of transaction analysis as envisioned by Bell Telephone of Canada. They were very religious in the use of the structure as a distinct four-level structure with no exceptions. This unnecessary restriction caused the philosophy to break down to the point of being unworkable. It is believed that the basic structure is a valuable starting point, but it is important to realize that consolidation of the structure may be possible. Consolidation in this context means that a module may be trivial enough to fit in its superordinate. Do not be afraid to modify the structure.

At this point, the obvious question is "But hasn't this design strategy created an uncohesive module at the top?" The answer would have to be yes most of the time. Before trying to increase the strength of the module though, consider the fact that the design represents the problem. If the design does represent the problem, then why obscure the problem in an attempt to increase module strength. If the solution is a good interpretation of the problem, then it may be best to accept a weak module. Most important, the design has been carefully analyzed and it can now be defended as a true representation of the problem.

QUESTIONS/EXERCISES

1. State in your own words what transform analysis is.

2. What is the function of the central transform?

3. The data identifying the termination point of an input stream is known as the most abstract form of the input. How is the output stream identified?

4. What is the primary design tool used in transform analysis? What other design tools are used?

5. List the four steps of transform analysis.

6. List the four management parallels used as guidelines for organizing the solution.

7. Explain the concept of factoring. When is factoring complete? Why is it better to overfactor than underfactor?

8. Apply transform analysis to a problem with which you are thoroughly familiar. Remember to take time to look back at the original problem at each step in the factoring process.

9. Define module coupling.

10. List the characteristics which aid in determining module coupling and give a brief explanation of each.

11. Explain the difference between local and global parameters.

12. Using the complexity formula, calculate the complexity of interface for a common data region containing 25 variables and being used by three modules.

434

13. Define normal and pathological connections.

14. What is the best type of module communication? Why?

15. What is meant by module cohesion?

16. Briefly explain the six levels of cohesion.

428

17. What is the cohesion of "edit all data" and "calculate square root"?

18. What is the basic philosophy of transaction analysis?

19. Sketch the basic structure chart used in transaction analysis and label the transaction center as well as each level.

SECTION III

TOP-DOWN DESIGN IMPLEMENTATION

INTRODUCTION

In the previous section, the various design tools that are available for your use were discussed. This section covers the techniques for implementing the top-down design: Top-Down Implementation, Program Design Language (PDL), and Structured Code. Top-down implementation is an approach to the design, coding, and testing of the system. Program Design Language will facilitate the translation of the design into Structured Code which is the final step in the translation.

CHAPTER 9

TOP-DOWN IMPLEMENTATION STRATEGY

When you have completed this chapter, you will understand the steps required for top-down implementation and be able to identify the associated terminology.

INTRODUCTION

First, let's talk about what the term implementation means. Some confusion is inherent since there are different levels of implementation. The implementation process begins when the system analyst develops a rudimentary design and assigns the problem to individuals/team. The second level of implementation is the detailed design, coding, and testing by the individuals/team, while the third level is the user's implementation of the program on his system. The Air Force definition of Top-Down Implementation refers to the second level which is often called the programmer's level. Then top-down implementation of modules will include coding and verification as well as implementation in a top-down fashion. That is, the higher levels of system logic are completed prior to subordinate modules.

The upper level modules are completely coded, tested, and all interfaces are fully identified before the coding of subordinate level modules is started. This is considered to be the best approach to coding implementation.

The other aspect of top-down implementation that this chapter will address is "packaging" the system prior to beginning the detailed design.

PACKAGING

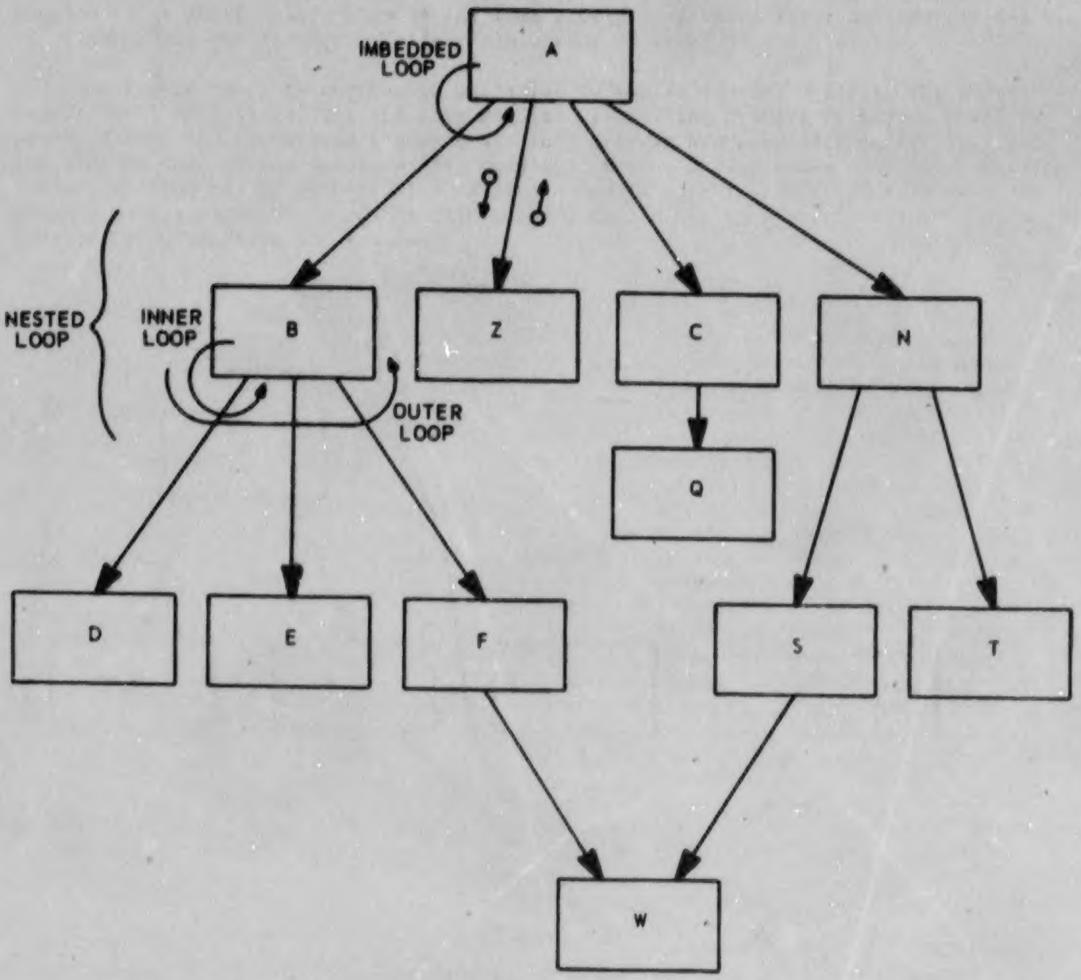
Packaging (which is also called "Builds," "Incremental Delivery," and "Delivery by Parts") is a tool that is used to determine where to divide the problem for development. It is the "art" of subdividing a skeletal design into several parts or packages and is usually accomplished by the chief programmer or his equivalent. A system can be subdivided for assignment to teams for detailed development; a package can show which group of modules to code first; or a package can be the parts of a system that will be delivered together if the user wants a delivery in increments. In any case, packaging should be delayed until the skeletal structure is as detailed as possible. (This is partly because it obscures the basic nature of the problem, and partly because it leads to gross inefficiencies if accomplished too early.) This will help you to more accurately choose the parts of the project that should be developed together.

Here are some suggestions for packaging for additional program efficiency. Modules that should be grouped together can be identified by looking for iterations (looping), communication, interval, and fan-in. Note that packaging is used to determine which modules will be grouped together for development. It does not determine the priority for which modules will be developed first.

The facts that were considered when packaging the program in figure 9-1 are listed as noted in figure 9-1. The reasons for this specific packaging is included in figure 9-2.

Within each package, the top module will be developed first--Top-Down!

In the case of iterations, put the module containing the loop and the subordinate modules that are called within the loop into the same physical package. Keep in mind that inner loops should take priority over outer loops, and nested loops should take priority over imbedded loops. (See figure 9-1.)

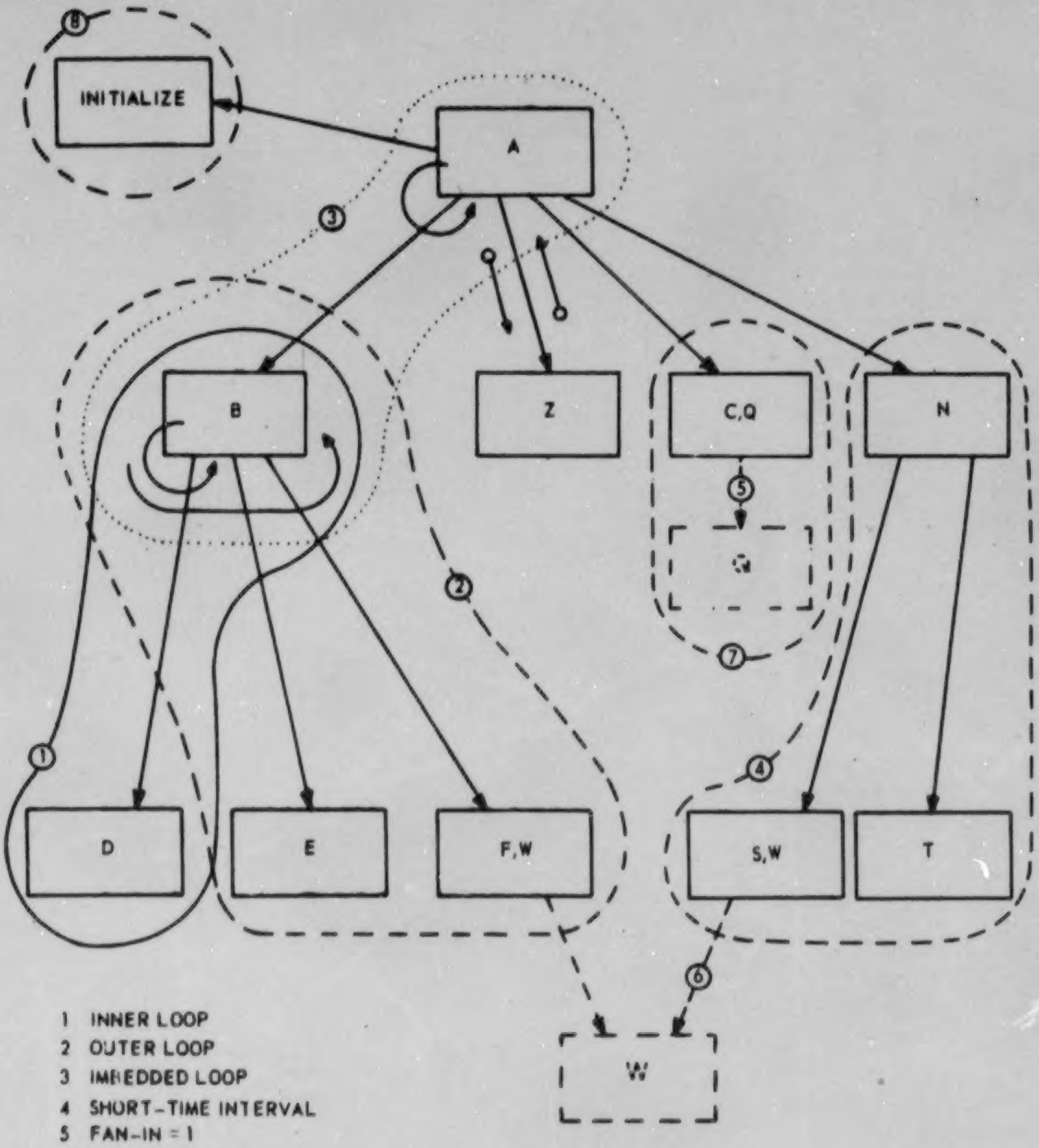


NOTES:

1. Module C called only on hardware failure.
2. Module T called 15 msec after Module S; all other modules called at intervals of 45 msec to 1 sec.
3. Module Z input: XYLIST, Plane, EOF, XCOORD, Error
   output: Error, IFLAG, YCOORD, Angle

432

Figure 9-1



1 INNER LOOP
2 OUTER LOOP
3 IMBEDDED LOOP
4 SHORT-TIME INTERVAL
5 FAN-IN = 1
6 FAN-IN SMALL
7 SMALL CHANCE OF EXECUTION
8 USED ONLY ONCE

NOTE: MODULES A AND Z ARE NOT PACKAGED TOGETHER
BECAUSE THE IMBEDDED LOOP CALLS B.

Figure 9-2

A high volume of intercommunication between two modules would make those modules logical candidates for incorporation into a single package. Communication volume is determined by the number of data items or control flags being passed between modules.

After looking at iterations and communication volume when packaging, time interval should be considered next. When you expect the time interval between the use of two modules to be short, place them in the same package. A short interval implies that the function of the two modules is either similar or at least closely related.

The fourth thing to check when packaging is module fan-in. This is the number of higher level modules calling the same module. If all the modules of a fan-in are not packaged into one group (see figure 9-3), then drivers are necessary to simulate the function of some of the superordinate modules. When a module calls only one subordinate module, the two may be combined into a single module rather than being left separate and grouped into a package. A module with a small fan-in can be compressed into the calling modules by duplicating in-line code.

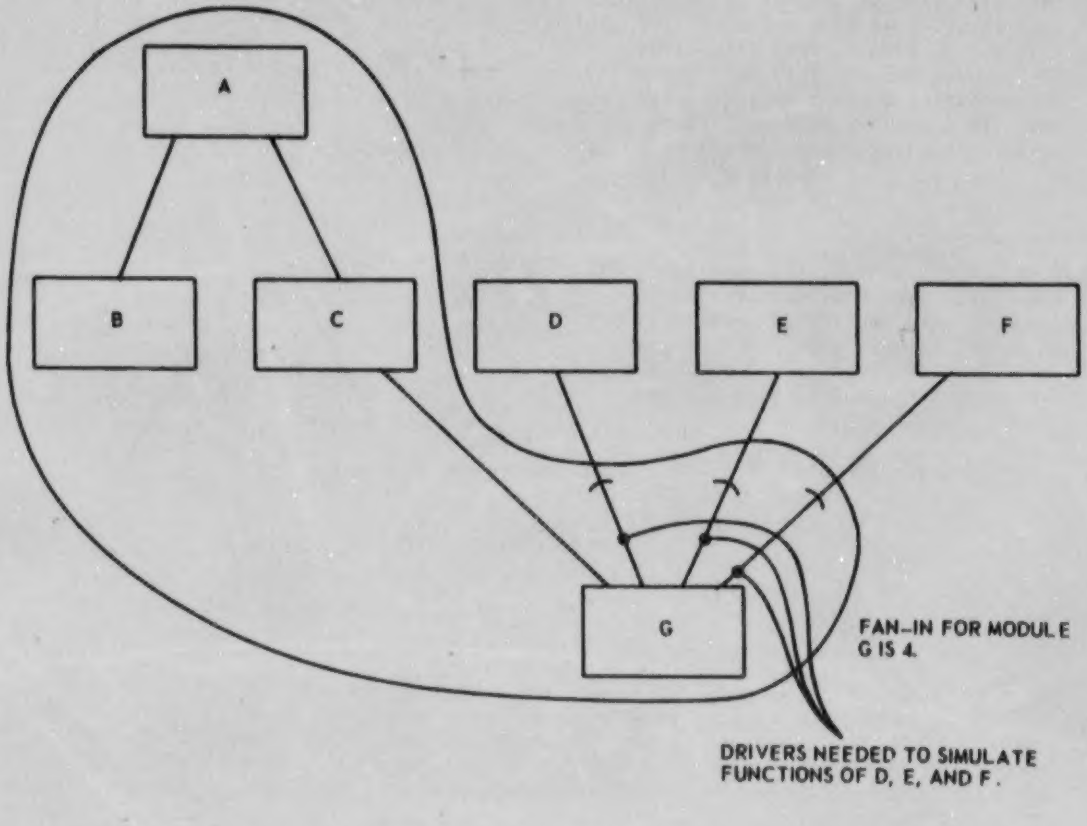

A
B
C
D
E
F
G
FAN-IN FOR MODULE G IS 4.
DRIVERS NEEDED TO SIMULATE FUNCTIONS OF D, E, AND F.

Figure 9-3

Some modules may be isolated into separate packages. A module that has only a small probability of being executed may be put into a separate package for development at a later time. A module used only once (such as initialization or termination) may be isolated. Two modules separated by a long processing interval could be placed in two separate packages.

A system analyst often begins by packaging a system (or program), followed by the flowchart, and then the design (hopefully structured design). This approach can cause problems for the programmer and can waste machine time and core. The proper approach would be to do a skeletal structured design, use the design to divide the system into packages, and then finish the packages using the tools available to him (flowcharts, HIPO, etc.). If the CPT concept is being used, the packages would be assigned to teams for individual development.

Again, packaging is just a way of grouping parts of a system into logical subparts for detailed development. The forming of these packages is still an "art." It is not as important to remember the different terms as it is to remember the concept and to use it to your advantage on the job.

Once the system is packaged, the next step is coding implementation which is normally accomplished by the programmer. In talking about coding implementation, regardless of whether you are responsible for an entire program or only one package, the same principle will apply.

CODING IMPLEMENTATION

While traditional software design may have been accomplished in a top-down manner, the actual software coding and testing has normally been done in a bottom-up fashion. When using the bottom-up approach, the lowest level processing modules are coded and unit tested first. Throwaway code, in the form of driver modules, is usually needed to perform the testing of the lower modules and these lower level modules cannot be integrated into the system since the higher-level, more-important, modules have not been coded.

Another problem is that data definitions and interfaces tend to be interpreted differently by each programmer involved and the differences are not discovered until the system test and evaluation phase. The system test must be delayed until the problems are resolved. Then, the problem solution usually requires modification of some, if not all, of the individual modules. Any individual module that is changed must be unit tested again before it can be integrated into the system.

Remember, all this modifying, redefining, and retesting is done during the system tests when you are desperately trying to meet your delivery date. Top-down coding implementation will eliminate most - if not all - of these problems.

In top-down, the highest level module is coded first and in complete form. Any CALLS, etc., that are required are left unresolved. All interfaces to the next lower level code are defined in complete detail. When the high-level module is coded, it is then tested. To meet the requirement for those lower level modules that would be CALLed, dummy code (called program stubs) is written. This throwaway code may involve nothing more than a message to a printer data set that "MODULE (name) HAS BEEN ENTERED." In other words, this module was successfully reached during execution. If necessary the stub can return a hard-coded parameter list, condition/completion code, or table of data - depending on what the highest level module interface would normally expect. If several stubs are necessary, frequently the same stub with several aliases will suffice.

Figure 9-4 shows the progress, top-down, which takes place when top-down development and implementation concepts are followed.

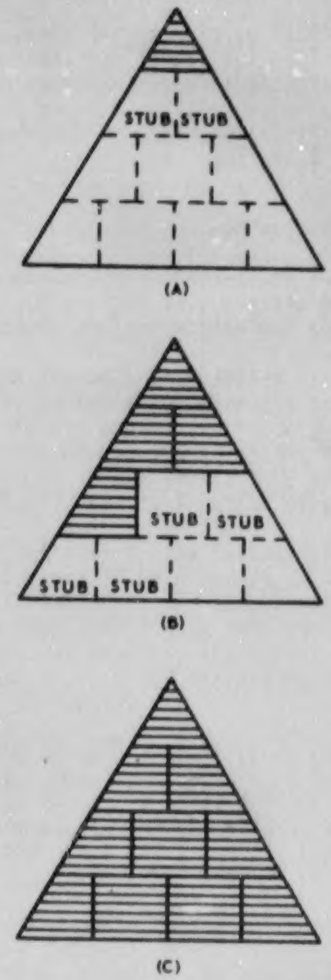


Figure 9-4

(a) Start of coding. Coding begins with the highest level of tasks in the program. What is the task of the program as a whole? It is defined in this section of coding. This highest level may only list the subtasks to be performed in each subsidiary section on lower levels.

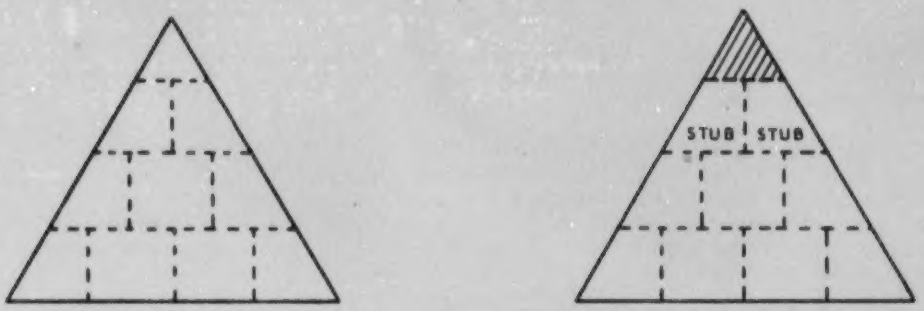
(b) Middle of coding. Subtasks are now being coded, and they may also be tested and integrated into the part of the program which is already complete, using the lower level "stubs" to test the logic in the completed portions. The stubs are used to indicate when a particular subtask section (to be coded) has been reached successfully. Reaching that section indicates that subtask has been done, checking out all of the logical paths in the completed portions.

(c) Completed coding. The program is complete and ready now for final testing, if tests have already been done at all of the higher levels. Tests for the last level coded can, in fact, be that part of the final acceptance tests.

There are three basic approaches to implementing top-down coding. Each of the three can be a valid approach under certain circumstances.

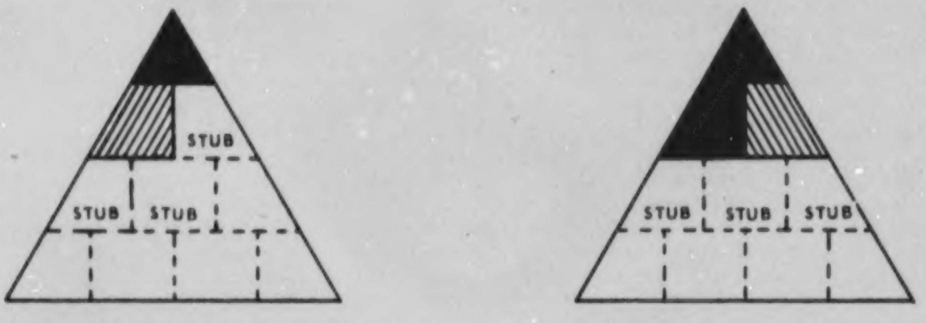
The preferred approach to top-down coding implementation is as follows: Start with a rough design of the whole system, complete the top-level detailed design, code and test that level. Next, design one module on the next lower level, code it, and test it with the previous module. As a new module is completed, integrate it with all previous modules, testing all modules each time until you finish the entire project. (See figure 9-5.)

Most important, the program modules at each level are fully verified and integrated with their predecessors before coding begins on the next lower level.



ROUGH DESIGN

DESIGN, CODE, TEST 1ST LEVEL



DESIGN, CODE, TEST & INTEGRATE ONE STUB

DESIGN, CODE, TEST & INTEGRATE ANOTHER STUB

Figure 9-5

An alternative approach that might work best for a very small project is to design the whole system top-down, code all of it top-down, and then test the completed system.

The third approach is to finish the entire design and then code and test the modules one at a time, top-down. This method could also apply to small systems and is actually preferred over the method that does all testing after coding the entire system.

Regardless of which one of these approaches you use, you will produce a much better program.

ADVANTAGES OF TOP-DOWN IMPLEMENTATION

You have seen packaging and top-down coding described as tools for implementing programs and you may be wondering how all of this can help you. There are several ways that top-down implementation can help.

There is continual testing of the interface between modules. As each module is completed, it is tested with all previously written modules. This checks the interface between the new module and the previous modules. If there are any interface errors, they are discovered and corrected early. Once the module has been integrated into the structure, it is not likely that interface problems will occur in the upper levels of the program as the development progresses.

Another time-saving feature of top-down implementation is that system integration is taking place as each module is tested and debugged. This has the effect of eliminating the system test and integration phase at the end of the project. Traditionally, test and integration have consumed an unreasonable amount of machine time and often the programmer has found it impossible to shoehorn his modules into a workable program. Figure 9-6 depicts the difference between the integration of a top-down program and a traditional program.

Along the lines of testing, the most important things are tested first. The higher level logic which contains the overall program logic is coded and tested prior to the details in the lower level modules being coded. This type of testing leads to a smoother evolution of the program with fewer surprises and less rewriting. Coding and testing the upper level modules first eliminates the need for special modules (drivers) to test the lower level modules.

It may seem at this point that all of the advantages of top-down implementation are related to integration and testing. Not true. Top-down implementation gives us a usable skeleton early. As succeeding lower levels are coded, tested, and integrated, we can decide to complete some parts of the system before other parts; thereby, making a portion of the system available to the user early. If the user should decide that he wants a partial delivery, he must be willing to accept this version and its associated restrictions.

The final advantage to top-down implementation is actually a side effect of all of the top-down tools and methods. Documentation is no longer an after-the-fact job. As the new modules are integrated, the documentation for each new module is added to the documentation package. This continual updating of the documentation reduces the time required to finish the documentation package at the end of the project. Plus, there is an added benefit. If someone must be replaced in an emergency, the replacement can readily see what has been done and what still needs to be done.

445

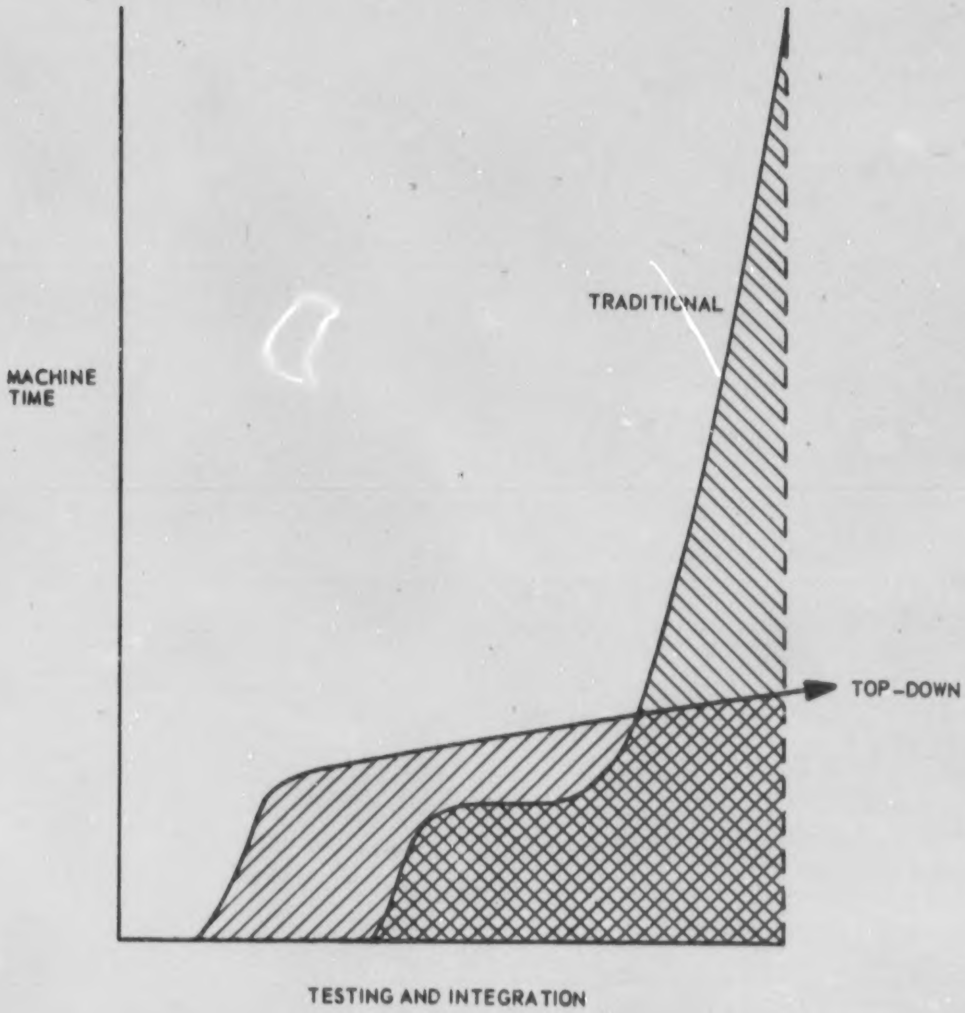


Figure 9-6

DISADVANTAGES OF TOP-DOWN IMPLEMENTATION

Top-down implementation offers the programmer, the manager, and the user many advantages. But, all is not so good in the land of top-down. The old saying "nothing is perfect" can also apply to top-down implementation. Some problems will appear.

First, the program design must allow top-down implementation. That is to say, the program must be designed in a top-down manner before top-down implementation can be employed.

Second, top-down requires the use of stubs and they may be throwaway code. You learned that drivers were throwaway code and should be avoided. The difference between drivers and stubs is that stubs may be written more easily and may be written in a way that makes them usable when the stub is coded. This is done by making the stub a simplified version of the function that it simulates and then using the same code in the final version.

Another problem arises when there are complex or critical lower level modules that need to be coded and tested immediately. When this happens, you should code only those lower level modules that are absolutely necessary and then return to the top-down development as soon as possible.

Other problem areas are: programs that are too small for top-down development; some problems do not lend themselves to top-down design/implementation (mathematical routines or I/O routines). Finally, when trouble is encountered in upper level routines, lower level design, coding, and testing is delayed.

Figure 9-7 summarizes the advantages and disadvantages of top-down implementation. Examine the figure carefully before reading the explanation on bottom-up implementation.

TOP-DOWN IMPLEMENTATION

Advantages

(1) Continual testing of interfaces between modules
(2) Earlier system integration
(3) Most important things tested first
(4) Usable skeleton early
(5) Documentation developed systematically

Disadvantages

(1) Design must allow top-down implementation
(2) Requires the use of stubs
(3) Complex or critical lower level modules may need early attention
(4) Programs too small for top-down
(5) Some problems do not lend themselves to top-down
(6) Lower level work may be delayed

BOTTOM-UP IMPLEMENTATION

Advantages

(1) Independent module testing
(2) Errors easily localized

Disadvantages

(1) Large area of unknown interfaces
(2) Inability to package bottom-up systems

Figure 9-7

BOTTOM-UP IMPLEMENTATION

Deviation from top-down normally occurs when an EXECUTE or CALL is encountered. The natural tendency when one writes this type of statement is to code the entire subordinate task before continuing the coding of the top-level module. When coding is done in execution sequence, it is called the "Bottom-Up" approach. The lowest level module is coded and completed before the intermediate level modules and the last module to be completed is the highest level module. "Driver" modules are often required because the upper level modules needed to supply data items to the lower level modules have not been coded. Frequently an entirely different driver is required for each unique module and the coding of drivers is time-consuming. Figure 9-8 is a graphic representation of a typical bottom-up development.

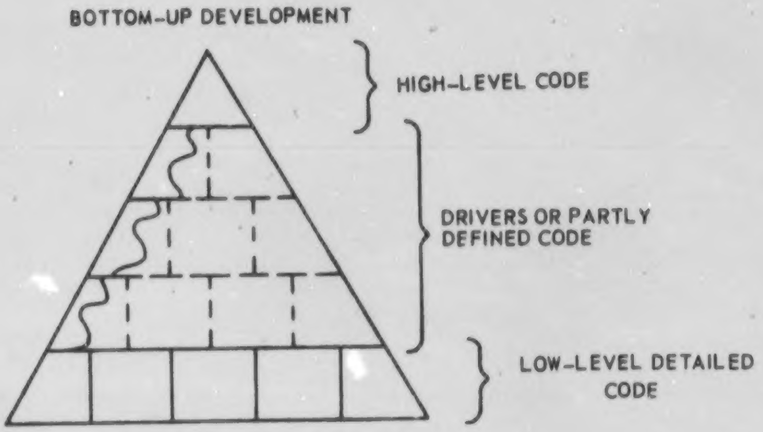


Figure 9-8

Disadvantages

There are two major disadvantages to the bottom-up approach. First, coding only part of the high-level modules and all of the lower level modules leave a large area in which many unknown interfaces exist. Since different programmers approach problems from different directions, even small decisions may have a large impact on how well the various modules fit together during integration. Because of programmer inconsistencies, a large amount of debugging time is consumed at integration time.

The second disadvantage of bottom-up is actually the reverse of one of the advantages of top-down. For years, software developers have searched for a way to deliver systems to users in a shorter period of time. When the top-down approach is followed, the system can be packaged and, therefore, delivered at an earlier date. With the bottom-up approach, the program is developed linearly. Obviously, linear development precludes packaging for parallel development of parts of the system.

Advantages

At the same time, there are two advantages to bottom-up implementation. The modules can be tested independently of the rest of the system - this allows a programmer to work autonomously and to avoid waiting for the rest of the system to be completed before he can test his module - and errors in logic within a module are easily localized. These two advantages seldom save enough to offset the accompanying problems when the system is integrated.

You have seen some of the advantages and disadvantages of bottom-up implementation. Compare these advantages and disadvantages with those for top-down implementation in figure 9-7.

Figure 9-9 shows the savings that can occur using top-down. The bottom-up approach allows very little overlap in the completion of project milestones. Design must be very near complete before work can begin on coding. Similarly, coding has to be quite complete before testing and integration of all of the program parts can begin. Only after all of the parts of the program are integrated can final testing begin. However, using the top-down approach, there is much more overlap. While the detailed, lower level subtasks are being designed, coding can begin on the higher level tasks of the program, as these task definitions will be unaffected by the lower level tasks. Also, while coding is being completed on the lower level sections, testing and integration can begin on the higher level code already completed, using "stubs" to represent the lower levels. Similarly, final testing will have already begun when testing is begun on the lowest level coding in the program, because every level above that will already have been tested. Also, the final testing phase will surely be shortened because of the previous testing at higher levels.

The next two chapters in this section will provide you with the final two tools you will need to complete the top-down implementation of the system. Program Design Language and Structured Code are the tools that will convert the design into a machine readable language.

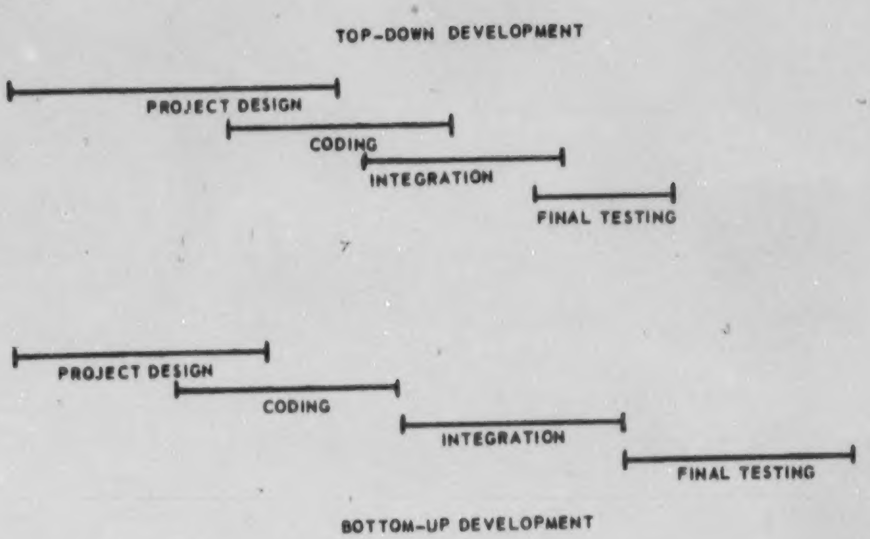


Figure 9-9

REVIEW QUESTIONS

Read and answer the review questions below.

1. Match the listed items.

Stub ____________

Bottom-up _______

Top-down _______

Driver __________

a. The lowest level modules are coded first.

b. An overall strategy for system development.

c. Used when required higher level modules are incomplete.

d. Used to indicate attempted use of a lower level module.

2. Which of the following is/are advantages of the top-down method?
   - a. Interfaces are resolved at the highest level.
   - b. Integration takes place concurrently with testing.
   - c. Drivers need not be written.
   - d. It does not take as long to design a system.
   - e. Undefined or hypothetical interfaces exist.
   - f. Stubs may be general and reusable.

3. One advantage of bottom-up implementation is that
   - a. interface problems do not have to be considered until later, thus allowing more time to concentrate on lower level modules.
   - b. one driver can be used to test all aspects of a module in a single run, thus checking out the module's interface.
   - c. rewrites of higher level modules may be necessary after interfacing of modules.
   - d. None of the above.

4. The following phrases are advantages of either top-down or bottom-up implementation. Enter the correct method in the spaces provided.
   - a. Independent module testing ____________
   - b. Earlier system integration ____________
   - c. Most important things tested first ____________
   - d. Documentation developed systematically ____________
   - e. Errors easily localized ____________
   - f. Usable skeleton early ____________
   - g. Continual testing of interface between modules ____________

5. The Air Force definition of top-down implementation is directed at
   - a. assignment of tasks at the beginning of the project.
   - b. the method for actually developing the program.
   - c. implementation of the completed system by the user.
   - d. All of the above.

6. The important point about top-down implementation is that
   - a. program modules are fully integrated and verified with their predecessors before coding begins on the next lower level.
   - b. program modules are tested independently of each other.
   - c. stubs are throwaway code also.
   - d. It is new and therefore must be good.

CHAPTER 10

PROGRAM DESIGN LANGUAGE

Upon completion of this chapter, you should know:

1. The purpose of Program Design Language (PDL).
2. The advantages and disadvantages of PDL.
3. The difference between freeform and formalized PDL.
4. The recommended language structures.
5. How to determine the level of PDL required in given situations.

INTRODUCTION

The use of structured programming technology has created a need to assess and improve the design tools which are used to prepare top-down structured programs. As you know, the flowchart has traditionally been the primary program design document. They have been used to graphically document, partially or totally, the design of a program before it is written in the target programming language. In many cases, however, flowcharts were produced after the fact in order to satisfy a documentation requirement. Top-down structured programming has created the need for new techniques to support the development process. This chapter defines and describes one such technique - a program design language.

DEFINITION

A Program Design Langugage (PDL) is a language for describing the control structure and general organization of a computer program. It is an English-like representation of a procedure which is easy to read and comprehend. It is structured in the sense that it utilizes the predefined control logic primitives. Indentation is used to make the PDL easier to read. This technique facilitates the translation of functional specifications into computer instructions using top-down design and structured coding. Figure 10-1 is an example of a PDL using structured control figures and indentation to represent hierarchical levels.

```
WORD FREQUENCY ANALYSIS PROGRAM

INITIALIZE THE PROGRAM
READ THE FIRST TEXT RECORD
DO WHILE THERE ARE MORE TEXT RECORDS
. . DO WHILE THERE ARE MORE WORDS IN THE TEXT RECORD
. . . . EXTRACT THE NEXT TEXT WORD
. . . . SEARCH THE WORD-TABLE FOR THE EXTRACTED WORD
. . . . IF THE EXTRACTED WORD IS FOUND
. . . . . . INCREMENT THE WORD'S OCCURRENCE COUNT
. . . . ELSE
. . . . . . INSERT THE EXTRACTED WORD INTO THE TABLE
. . . . END IF
. . . . INCREMENT THE WORDS-PROCESSED COUNT
. . END DO AT THE END OF THE TEXT RECORD
END DO WHEN ALL TEXT RECORDS HAVE BEEN READ
PRINT THE TABLE AND SUMMARY INFORMATION
TERMINATE THE PROGRAM
```

Figure 10-1. Program Design Language (PDL) Example

PURPOSE

The primary purpose of this technique is to help the programmer translate the functional specifications into computer instructions using top-down structured programming. The PDL is a multipurpose design tool, which has been used in the design, development, documentation, and maintenance of structured programs.

A PDL assists the design process because it can be used to develop and study alternate control structures and general program organizations easily and at a relatively low cost. Most designs require review and rework either to correct errors or to improve efficiency. A PDL is also an excellent medium to implement a review of design alternatives. The use of PDL for design verification and code verification is discussed in "Validation and Verification Study," Volume XV of the RADC Structured Programming Series.

The use of PDL for documentation is discussed in "Documentation Standards," Volume VII of the RADC Structured Programming Series. In the past, flowcharts have been required as part of the documentation accompanying program specifications. This practice should be discouraged because structured programming technology eliminates the need for a solution to be depicted in the programming specifications before coding begins. In top-down programming, there should be an overlap of design and implementation activities; hence, there is no requirement to show the proposed solution in the specification in either flowchart or PDL form.

A PDL assists program maintenance (both corrections and modifications) when it is used to express complicated algorithms, algorithms critical to performance, and programs which are difficult to read. In some instances, PDL has been used as documentation for assembly language programs. If a structured source code listing of a high level language is available, then the PDL representation is redundant and should not be required. Volume VII of the RADC Structured Programming Series discusses this point in more detail.

ADVANTAGES OF USING PDL

Some of the reasons for using PDL are:

1. PDL is easier to prepare when compared with graphical methods such as detailed design flowcharts.

2. It is easier to read and comprehend as compared to flowcharts or long involved prose descriptions.

3. It facilitates the translation of design into a top-down structured program.

Preparation

The preparation and publication of PDL representations are simpler processes than the preparation and production of flowcharts. Since PDL does not use special symbols, it can be developed and produced without the use of templates. Therefore, it can make effective use of all the technology which has been developed for the production of written material (e.g., typewriters, computer-based text editing systems). The PDL source text can (and should) be stored and maintained using the same programming support library system used for the structured programs. The machine readable form of PDL would be available for efficient transmission over communication facilities. The ease of preparation makes the generation of PDL by programmers both fast and relatively inexpensive.

In addition to the ease of initial production, the ease of maintenance of PDL makes PDL representations more useful than flowcharts. It is easier, less time-consuming, and less tedious to change a PDL representation than to redraw a flowchart. For instance, it is not uncommon during program development to be faced with the requirement of having to add one more block to an already prepared flowchart. If the addition cannot be easily added to the flowchart, one of the following alternatives is taken:

1. The flowchart is not updated.

2. The flowchart is redrawn to add the additional logic.

3. The flowchart is patched with off-page connectors and a separate page containing the new logic is added.

The acceptance of the first choice will result in an incomplete, obsolete, and often erroneous document which loses its creditability and utility. The contrast is the easy update of a PDL segment.

Usability

PDL representations have been found to be very readable and comprehendable. The absence of graphic symbols makes it much easier to use meaningful naming conventions since the programmer is no longer constrained by graphic symbols.

The characters 'FICA LT' fit in the standard decision block but 'IF FICA LIMIT VALUE', which is much more meaningful, will not. A PDL writer can choose names which precisely represent the action or device he wishes to describe (e.g., CARD, DRUM, DISK).

A program design in PDL is a precise image of the program implementation in a top-down structured programming environment. The top-level segment summarizes the general program organization. Detailed decisions in lower level segments are not represented until the segment itself is expressed in PDL. Figure 10-2 contains a PDL representation of a top-level code segment.

```
BEGIN PROGRAM
   DO WHILE INPUT CARDS EXIST
      IF TRANSACTION CODE = TYPE 1 THEN
         INCLUDE A
      ELSE
         INCLUDE B
      ENDIF
   ENDDO
END PROGRAM
```

Figure 10-2. Top-Level Code Segment

This simple PDL representation of a top-level segment shows that two subordinate segments will be utilized to handle the different transaction types. What is important is that this does show the precise organization of the program. The two subsegments are in fact segments in the final code. Low-level decisions which are in either "A" or "B" will only be shown in the expansion of "A" or "B." This example also illustrates that PDL utilizes the indentation scheme used in structured programming to show nested logic enhancing PDL readability. The PDL keywords (e.g., IF, DO WHILE, INCLUDE) can either be chosen by the writer or can be predefined in the PDL language specification itself.

Promotion of Top-Down Structured Programming

The translation from design to structured programs is relatively simple with PDL. Since PDL uses the structured programming figures along with indentation conventions, it does not require the programmer to first interpret graphical flowcharting symbols and translate them into code. Figure 10-3 provides an example. It contains two representations of the same procedure: a flowchart and a PDL.

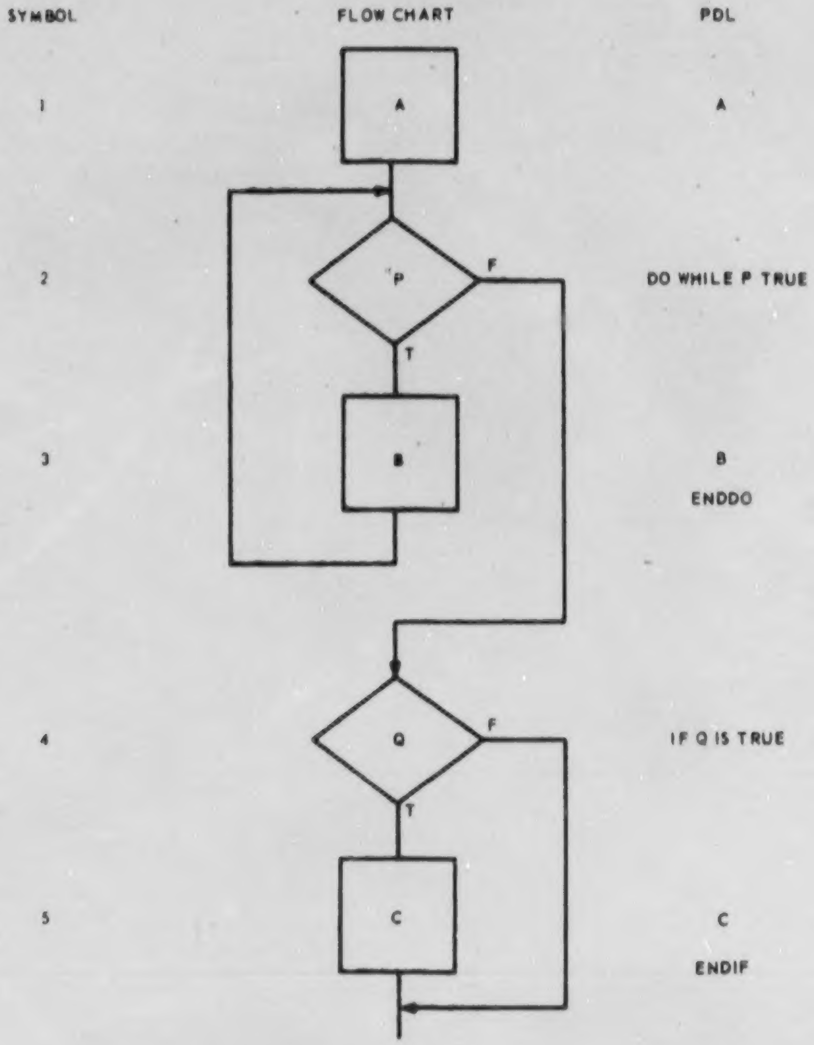


Figure 10-3. Two Representations of the Same Procedure

Using the flowchart requires the programmer to correctly interpret that symbols 2 and 3 are grouped together to form the DO WHILE and that symbols 4 and 5 form an IFTHENELSE. In a large problem, this type of interpretation can be difficult and often leads to invalid programs. As shown, the PDL is already in the proper form. No interpretation is needed; however, translation of PDL to the target code is still required. PDL should be used in conjunction with structured programming because of the indescriminate branching in unstructured code.

The promotion of top-down structured programming is the most important advantage of PDL. This is due to the following:

1. PDL utilizes the same structured programming logic control figures coding basis as does top-down structured programming. (Refer to "Programming Language Standards," Volume I of the RADC Structured Programming Series.)

2. PDL uses indentation to facilitate readability, a major attribute of structured programs.

DISADVANTAGES OF USING PDL

Program design languages have disadvantages which include:

1. An attempt to replace an established and accepted design and document technique. Flowcharts are well-liked and used by many programmers. The transition from flowcharts to PDL is difficult as is any change.

2. Rules concerning the use of PDL. The rules must be followed or the advantages of PDL will not be realized.

3. The elimination of graphics. Graphic symbols contribute to the comprehension of the program design.

4. Will require time to learn the use and form of a PDL.

PROGRAM DESIGN LANGUAGE FORMS

Program design languages have been implemented and used with varying degrees of formality. Two forms, at each end of the spectrum, are discussed. The two forms are arbitrarily called Freeform and Formal.

Freeform PDL

DESCRIPTION: A freeform PDL is one which contains a minimal number of syntactical and semantic rules. A freeform PDL incorporates features structured programming, where the user uses natural language, control figures, indentation, and rules of segmentation in a structured English narrative. (See figure 10-4.)

The statement READ INCOME TAX RECORD is an example of a freeform PDL statement. In this example, READ is not a keyword of a specific PDL but an action verb chosen by the writer to indicate a general process of performing an input operation. The PDL writer could have expressed the same idea by writing MOVE INCOME TAX RECORD or TRANSFER INCOME TAX RECORD.

```
PRINT FICA REPORT HEADER
OBTAIN FICA PERCENT AND FICA LIMIT FROM CONSTRAINTS FILE
SET FICA TOTAL TO ZERO
DO FOR EACH RECORD IN SALARY FILE
   OBTAIN EMPLOYEE NUMBER AND TOTAL SALARY TO DATE
   IF TOTAL SALARY IS LESS THAN FICA LIMIT THEN
      SET FICA VALUE TO TOTAL SALARY TIMES FICA PERCENT
   ELSE
      SET FICA VALUE TO FICA LIMIT TIMES FICA PERCENT
   ENDIF
   PRINT EMPLOYEE NUMBER AND FICA VALUE
   ADD FICA VALUE TO FICA TOTAL
ENDDO
PRINT FICA TOTAL
```

Figure 10-4. Freeform Program Design Language (PDL) Example

Another example of a freeform PDL capability follows:

IF condition is valid THEN perform actions as shown in decision table 1.

ELSE perform actions as shown in decision table 2.

END IF

This example shows how decision tables could be referenced in the freeform PDL by writing an action statement.

A freeform PDL is easy to learn and use. The absence of rigid syntax and semantic definition eliminates the necessity of learning a precise syntax, the equivalent of learning another programming language. This absence facilitates the writing by programmers and the understanding by nonprogramming personnel (e.g., managers and analysts). A freeform PDL is versatile and can be used as a design aid to assist a programmer in designing a program in any programming language. Since the syntax and semantic definitions will be defined by individual PDL users, the consistency among various PDL users will vary; but, as long as all the users adhere to the principles of top-down structured programming, the logic represented by the PDL will be consistent even though the form may vary slightly.

Formal PDL

A formal PDL is one which contains precise syntactical and semantic definitions. It incorporates the principles of structured programming, by defining rigid keywords (e.g., DO WHILE, IF, THEN, ELSE) and additional keywords for other operations (e.g., READ, WRITE, SET). Rules for correct syntactical constructions are also included. An example of a formal PDL statement is:

READ (INCOME-TAX-RECORD) INTO AREA

In this example, READ and INTO are specific keywords of the language. The syntactical requirement is that they are underlined, that the record name be in parenthesis, and that the record name be a contiguous set of characters.

Another example of a formal PDL is:

```
IF CONDITION A THEN
   DO
      REFERENCE DECISION TABLE (Name)
      PERFORM (ACTIONS) AS SPECIFIED
   ENDDO
ENDIF
```

REFERENCE, DECISION TABLE, and PERFORM, as specified, are keywords in the PDL. The decision table would probably have a specific format and structure. This approach requires that a set of keywords be chosen and defined in the PDL. All users of the PDL would be required to use the same conventions.

The degree of formality of a PDL is not rigid; rather, it is variable depending on the developer of the PDL. As an example, additional items that could be included in a formal PDL description include:

1. Data descriptions - The type of data allowable by this program (e.g., character, integer).

2. Computational descriptions - The precise definition and order of algebraic manipulations (e.g., +, -, /, *).

A formal PDL is more precise than a freeform PDL. The standardization and addition of syntactical and semantic definitions would assure that all users of the PDL be more consistent in improving the readability and understandability of each other's PDL representations. The argument for the more precise syntax (e.g., READ, GET, PROCESS) and semantics is that the preciseness provides better communications between the various users of the PDL (e.g., the designers and coders) since there are fewer ambiguities. Whether or not the amount of improvement would be worth the time and effort is not known at this time.

A formal PDL is more difficult to learn and use than a freeform PDL. The syntactical and semantic definitions must be learned, a process which could be equivalent to learning another programming language. This difficulty of learning a formal PDL might hinder its use. Another concern is that the meaning of what is to be represented in the PDL might be lost in the rules of the language which attempt to describe it.

A formally defined PDL could be processed by a computer. The PDL has syntax and semantic rules which could be analyzed by a computer analysis program. This computer analysis process could critique the design process. Possible errors, such as unused data items, data received too late for use, inconsistent use of data, are the types of problems that an analysis program might be able to identify.

Summary

The degree of formality of a PDL should depend on its use in various situations. A program design language used primarily as an aid to designers should probably be freeform while a program design language used primarily in formal documentation should be more formalized.

RECOMMENDED PDL

The recommended PDL has a low degree of formality and is based on the structured programming precompiler standards as established in "Programming Language Standards," Volume I of the RADC Structured Programming Series. This document which contains complete descriptions of the language structures should be used as the basic reference document.

The following are the recommended language structures with keywords capitalized and underlined.

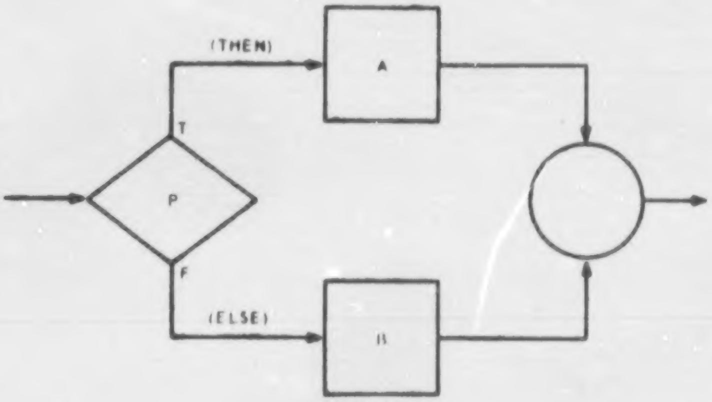
Sequence

Sequential statements, the most basic form of control flow, would have the following language text representation:

```
English Language
English Language
.
.
.
```

IFTHENELSE Figure

The IFTHENELSE figure causes control to be transferred to one of two functional blocks of code based on the evaluation of the truth of a conditional statement (p) acceptable to the source language. The flowchart for the IFTHENELSE figure is:

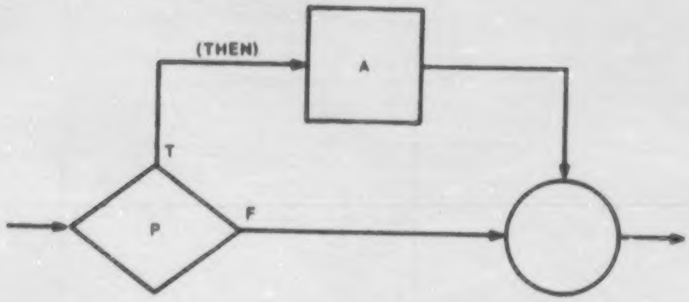


IFTHENELSE

and the language text to be used to represent the IFTHENELSE is:

```
IF (p) THEN
    English Language for A
ELSE
    English Language for B
ENDIF
```

The ELSE in the IFTHENELSE figure is regarded as optional; if the ELSE is omitted, the flowchart for this figure becomes:



IFTHEN

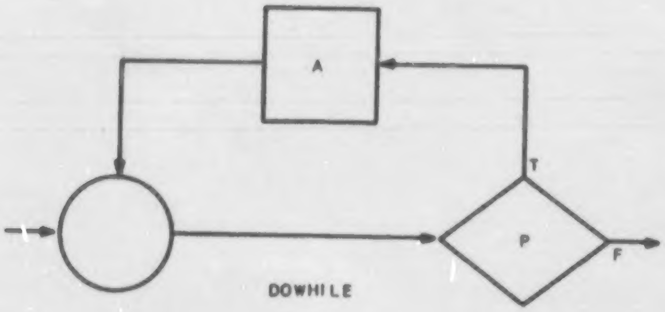
and the language text is:

```
IF (p) THEN
   English Language for A
ENDIF
```

The THEN is optional in both cases and if present is treated as a comment.

DOWHILE Figure

The DOWHILE figure allows execution of functional block of code A while a condition (p) is true. The flowchart for the DO WHILE figure is:



DOWHILE

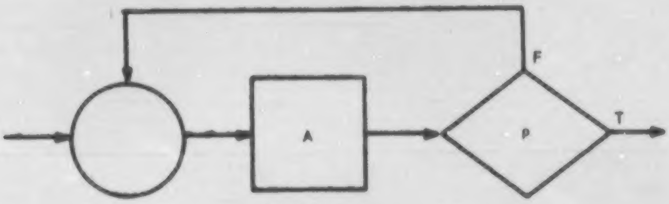
454

and the language text to be used is:

```
DO WHILE (p)
   English Language for A
ENDDO
```

DOUNTIL Figure

The DOUNTIL figure allows execution of a functional block of code A until a condition (p) becomes true. The functional block of code A is executed at least once. The flowchart for the DOUNTIL figure is:



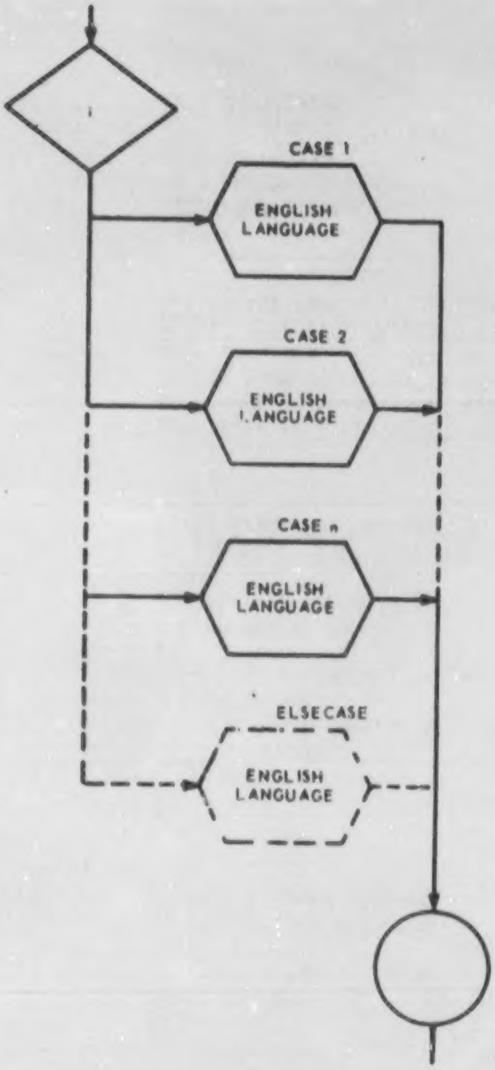
DOUNTIL

and the language text to use is:

```
DO UNTIL (p)                              DO English Language for A
   English Language for A    or             UNTIL (p)
ENDDO                                     ENDDO
```

CASE Figure

The CASE figure causes control to be passed to one of a set of functional blocks of code (A,B,...,Z) based on the value of an integer variable (i) = (1,2,...,m). The flowchart and language text for the CASE figure is:



```
CASENTRY (i)

 CASE 1
     ENGLISH LANGUAGE

 CASE 2
     ENGLISH LANGUAGE

 CASE n
     ENGLISH LANGUAGE

 ELSECASE
     ENGLISH LANGUAGE

 ENDCASE
```

The default code (ELSECASE) is code which is to be executed if the value of i is outside of the limiting range of values of the indicated case numbers. This code (and its associated keyword) is optional and may be omitted. This action is different from the one where the number is within the indicated range of case numbers but no case exists for it. In this instance, control passes to the end of the CASE figure.

Summary

One generalized PDL based on the structured programming standards will stimulate the adherence to the standards and will make it useful for all the programming languages.

DEVELOPING LEVELS

You have seen the recommended Program Design Language (PDL) and are probably wondering how to do it. Like top-down design and implementation, PDL is developed one level at a time. When the word "level" is used, it not only refers to levels of modules (in design) but, also, to how "close" the PDL resembles the target languages. When writing, each step (or level) gets closer to actual code.

In starting the PDL (level 1), use simple English language statements. (Note: This process is done module by module if you are working from a structure chart or HIPO.) List the processes or functions that will be done in the order they are to be performed (see figure 10-5). Next, replace the functions with more descriptive primitives (level 2--see figure 10-6). After the level is written, analyze it to see if the functions will accomplish what is to be done. Then, repeat the last two steps as much as necessary (levels 3 and on--see figure 10-7). (The question as to how far to take this process is answered later. Let's first see how to do it.)

The following example will show the development of a PDL. The problem is converting a two-digit data item which represents the year of an unknown century to a four-digit data item for easier determination of when to downgrade classified material.

Level One - English Language Statements

```
FOUR-DIGIT-YEAR
INITIALIZE VARIABLES
CHECK INPUT DATA
CONVERT YEAR
END
```

Figure 10-5

Level Two - Nearer to Code

```
FOUR-DIGIT-YEAR
CALL INITIALIZE-VARIABLES
CALL CHECK-INPUT-DATA
CALL CONVERT-YEAR
END

INITIALIZE-VARIABLES ROUTINE
INITIALIZE VARIABLES AS NEEDED
END

CHECK-INPUT-DATA ROUTINE
CHECK TO SEE IF VARIABLES ARE WITHIN RANGE
END

CONVERT-YEAR ROUTINE
DETERMINE OUTPUT YEAR USING YEAR TAKEN
FROM COMPUTER AND YEAR RECEIVED FROM
FILE
END
```

Figure 10-6

Level Three - Still Nearer

```
      FOUR-DIGIT-YEAR
      CALL INITIALIZE-VARIABLES
      CALL CHECK-INPUT-DATA
      CALL CHECK-FOR-ERROR
      CALL CONVERT-YEAR
      END

      INITIALIZE-VARIABLES ROUTINE
      ACCEPT YEAR FROM COMPUTER SYSTEM
      MOVE THAT YEAR TO A VARIABLE, CURRENT-YEAR
      ACCEPT FROM FILE A VALUE FOR A VARIABLE, INPUT-YEAR
      MOVE ZERO TO A VARIABLE, ERROR-TYPE
      MOVE ZERO TO A VARIABLE, OUTPUT-YEAR
      END

      CHECK-INPUT-DATA ROUTINE
      IF (CURRENT-YEAR NOT EQUAL TO THE ACTUAL YEAR)
      THEN
( 1)    MOVE 1 TO ERROR-TYPE
( 1)    CALL ERROR
      ELSE
( 1)    IF (INPUT-YEAR NOT NUMERIC)
( 1)    THEN
( 2)      MOVE 2 TO ERROR-TYPE
( 2)      CALL ERROR
( 1)    ENDIF
      ENDIF
      END

      CHECK-FOR-ERROR ROUTINE
      IF (ERROR-TYPE NOT = 0)
      THEN
( 1)    TRANSFER CONTROL TO PROGRAMMER
      ENDIF
      END

      CONVERT-YEAR ROUTINE
      IF (INPUT-YEAR < 17)
      THEN
( 1)    COMPUTE OUTPUT-YEAR = INPUT-YEAR + 2000
      ELSE
( 1)    IF (INPUT-YEAR - (CURRENT-YEAR - 60) > OR = 0)
( 1)    THEN
( 2)      COMPUTE OUTPUT-YEAR = INPUT-YEAR + 1900
( 1)    ELSE
( 2)      IF (INPUT-YEAR - (CURRENT-YEAR - 60) < 0)
( 2)      THEN
( 3)        COMPUTE OUTPUT-YEAR = INPUT-YEAR + 2000
```

Figure 10-7

```
( 2)        ENDIF
( 1)      ENDIF
        ENDIF
        IF (OUTPUT-YEAR < 1917 OR OUTPUT-YEAR > 2039)
        THEN
( 1)      MOVE 3 to ERROR-TYPE
( 1)      CALL ERROR
        ENDIF
        END
```

```
        ERROR ROUTINE
        CASENTRY (ERROR-TYPE)
        CASE (1)
( 1)      DISPLAY "CURRENT-YEAR IS INCORRECT"
        CASE (2)
( 1)      DISPLAY "INPUT-YEAR NOT NUMERIC"
        CASE (3)
( 1)      DISPLAY "OUTPUT-YEAR NOT WITHIN RANGE 1917 - 2039"
        ELSECASE
( 1)      DISPLAY "UNKNOWN ERROR"
        ENDCASE
        END
```

Figure 10-7 (continued)

You are now wondering "just how far do I take the PDL?" It depends upon just what the PDL is to be used for--overall design, detail design for coding, maintenance documentation, etc. An overall design PDL may only need to be done to the 2nd or 3rd level. The "coding level" should be that level which can easily be translated into code. (This level will vary somewhat from programmer to programmer.) If it is to be used for documentation (e.g., program description in a programmer's maintenance manual), perhaps 1 or 2 steps "back" from a "coding level." The main point is still, "What is the intended use of the PDL?"

If the above paragraph sounds like it is uncommitted, you are right! There is a definite reason for that. Determining the number of levels and deciding how to write PDL is going to come from practice and experience. It is just like starting a programming career. For the most part, a person is not "born" a programmer overnight. Likewise, a person is not "born" a PDL'er. Learning how to use PDL will take time.

A recommendation: Take a simple problem (or a module from a completed hierarchy chart, structure chart, or what have you) and use a PDL to complete a "solution." Start at the 1st level and work through the PDL until you feel that you can easily code from it (i.e., you feel that there is no ambiguity as to what code will be used to translate the PDL).

SUMMARY

A program design language has been defined as a language for describing the control structure and general organization of a computer program. Its primary purpose is to help the programmer in translating functional specifications into computer instructions.

Using a PDL has both advantages and disadvantages. Among the advantages are: (1) PDL is easier to prepare when compared with graphical methods; (2) it is easier to read and comprehend; and (3) it facilitates the translation of design into a top-down structured program. The disadvantages include: An attempt to replace an established and accepted design and document technique, rules concerning the use of PDL, the elimination of graphics, and has a learning curve.

The recommended PDL has a low degree of formality. The five basic language structure keywords are capitalized and underlined. The sequence, IFTHENELSE, DOWHILE, DOUNTIL, and CASE figures comprise the basic language structures recommended in the RADC Structured Programming Series.

The number of PDL levels that it takes to develop a program depends upon the intended purpose of the PDL.

STUDY QUESTIONS

1. A program design language is used

   a. to help translate functional specifications into computer code.

   b. to assist the design process.

   c. for describing the control structure and general organization of a computer program.

   d. All of the above.

2. The primary purpose of using PDL is

   a. to get rid of the "GO TO's."

   b. to help the programmer to translate the functional specifications into computer instructions.

3. Label the following as either an advantage or disadvantage for using a PDL:

   ____________ a. Easier to read and comprehend as compared to flowcharts or long involved prose descriptions.

   ____________ b. The rules concerning the use of PDL must be followed.

   ____________ c. Graphics are eliminated.

   ____________ d. It facilitates the translation of design into a top-down structured program.

4. The recommended PDL has a low degree of formality and

   a. the rules concerning its use should be followed so that the advantages can be realized.

   b. has the keywords capitalized and underlined.

   c. is based on the structured programming standards as established in Volume I of the RADC Structured Programming Series.

   d. All of the above.

5. The recommended language structures include:

   a. sequence, IFTHENELSE, DO FOREVER.

   b. sequence, IFTHENELSE, DOWHILE.

   c. DOUNTIL, CASE, SEQUENCE.

   d. b and c above.

6. How many levels is a PDL written to?

   a. 3

   b. As many as needed depending upon its intended use.

   c. Until it is "close" enough to easily write code from.

   d. None of the above.

Chapter 11

STRUCTURED CODING

INTRODUCTION

Structured programming, which includes structured coding, is often erroneously called "goto-less" programming. Although the use of GOTOs is minimized, some GOTOs are needed. A GOTO is only used to implement a specific control structure and then control should be transferred to another point within the same module. Some programmers object to structured programming because "it is difficult to do" or "it is too hard to understand." Actually, structured coding is not too difficult to do or to understand. There is a certain amount of learning associated with this new concept. Resistance to change and/or a lack of understanding of just what structured code is and what it can and cannot do causes the greatest problems.

Resistance to change must be overcome by each individual organization. The lack of understanding can be eliminated by educating the individual programmer. This chapter focuses on the latter aspect.

Upon completion of this chapter, you should be able to:

1. Define "structured coding."

2. Name the five basic control logic primitives and their related compiler language representations.

3. Define "precompiler."

4. Explain how a precompiler is utilized in structured programming.

You will also gain insight into possible problems encountered when using structured code and some solutions to these problems.

A comparison of "traditional code" and structured code is also presented in this chapter.

DEFINITION

Structured coding is defined as the coding of programs by repeated use of a selected number of predefined control logic primitives. These primitives (also called constructs) can be combined to implement a program design. The use of these primitives demands that simple code be written, thus increasing the readability and comprehension of program logic.

The control logic primitives are divided into three classical categories: sequential execution, conditional execution, and iterative execution. Each of these constructs must have only one entry point and one exit point. Usage of other primitives should be restricted to small, well-documented, and specialized functions which have been approved by management. Each of the three categories will be explained in this chapter.

Structured coding includes conventions for organizing code. For example:

1. Modules should be a "reasonable" size -- i.e.,
    a. less than 100 lines of coding.
    b. one printed page.
    c. two coding sheets.
2. Indent all code to clearly denote the logical levels of constructs.
3. Include embedded annotations to explain items not obvious in the code.
4. Group all format statements in one area to simplify maintenance and debugging.
5. Arrange all data in a meaningful order in contiguous areas.

CONTROL LOGIC PRIMITIVES

As stated earlier, there are three categories of primitives. The "sequential execution" is the execution of statements in a straight-line fashion with no branching or looping. The "conditional execution" is the same as the traditional branching statement. "Iterative execution" is another name for the looping process. These three categories of primitives form the basis for structured coding.

From these categories come the control logic primitives: sequence, IFTHENELSE, and DOWHILE. Two extensions of the categories are the DOUNTIL (iterative process) and CASENTRY (conditional branching).

The format, flow chart, actions taken, and compiler language code for each of the five primitives are illustrated. ANS COBOL, ANS FORTRAN, and J3 JOVIAL are used as representative compiler languages. Volumes I and II of the RADC Series contain more detailed information on these compiler languages and brief references to other languages (assembler, etc.).

Each primitive is illustrated in the following order in figure 11-1.

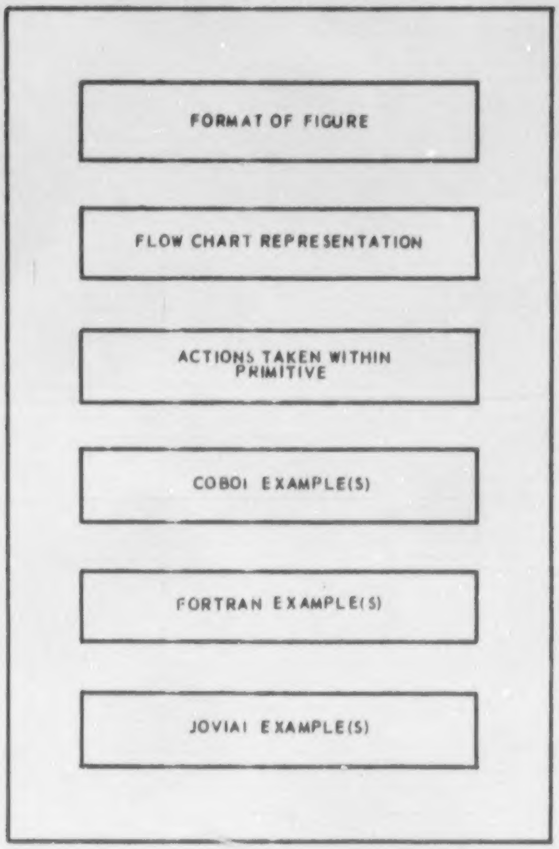


Figure 11-1

IFTHENELSE/IFTHEN Primitive

FORMAT

IFTHENELSE		IFTHEN
IF CONDITION-IS-TRUE		IF CONDITION-IS-TRUE
CODE A		CODE A
ELSE	OR	ENDIF
CODE B		CODE B
ENDIF		
CODE C		

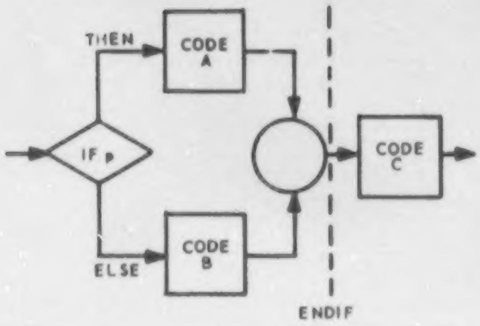


Figure 11-2A

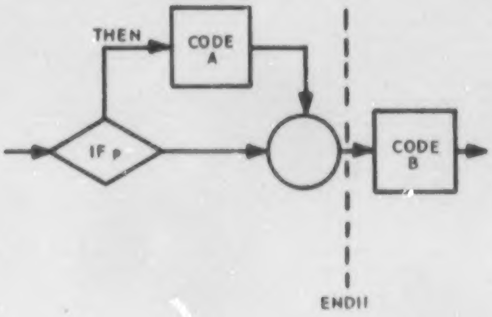


Figure 11-2B

When an IFTHENELSE is executed, the following actions are taken:

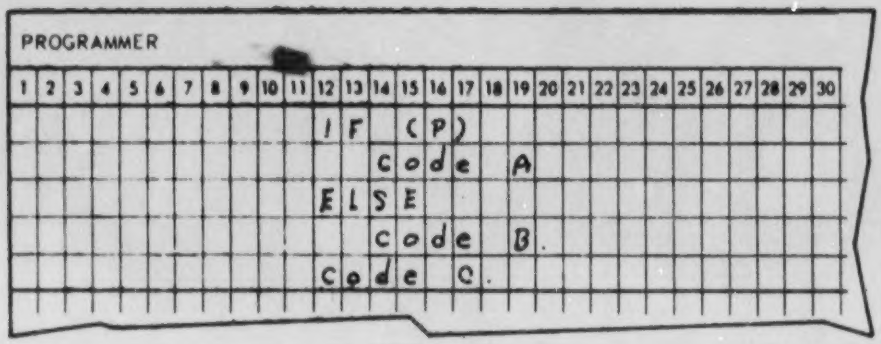
1. If condition p is true (figure 11-2A), code A is executed and control is passed to the statement following the ENDIF (code C in figure 11-2A and code B in figure 11-2b).

2. If condition p is false, code B following ELSE (figure 11-2A) is executed. When the ELSE option is omitted, as in figure 11-2B, the statement following the ENDIF is executed.

Code A and code B in the IFTHENELSE control structure figure may be comprised of one or more statements and/or control structure figures.

IFTHENELSE (Con't)

ANS COBOL


PROGRAMMER
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

```
           IF (P)
             code A
           ELSE
             code B.
           Code C.
```

Figure 11-3

Problems arise when nested IF statements are required in COBOL. Figure 11-5 illustrates three possible options to overcome the nesting limitations resulting from the following flow chart.

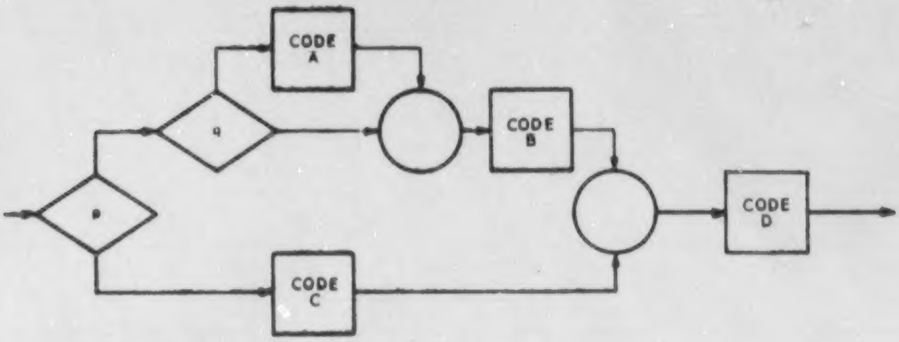
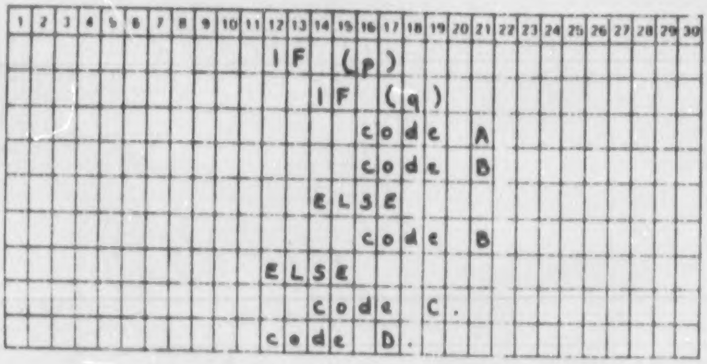


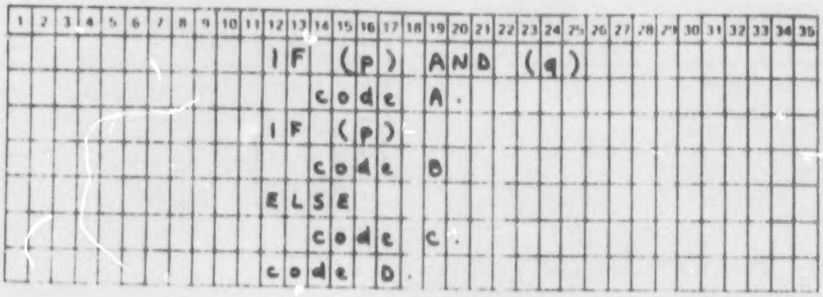
Figure 11-4

IFTHENELSE (Con't)



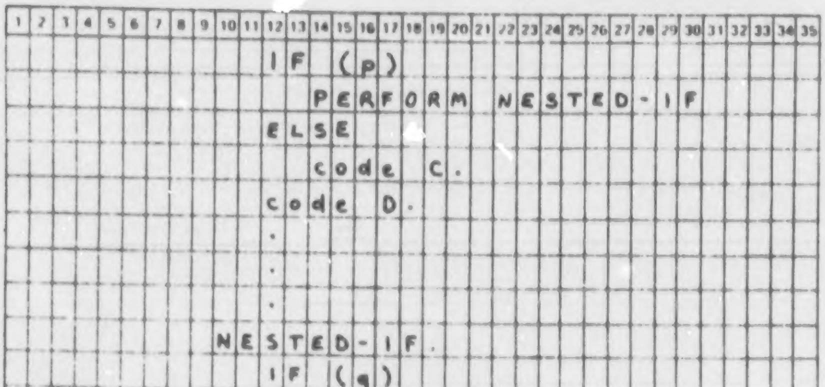
```
           IF (p)
             IF  (q)
               code A
               code B
             ELSE
               code B
           ELSE
             code C.
           code D.
```

(a) Duplicate code B



```
           IF (p) AND (q)
             code A.
           IF (p)
             code B
           ELSE
             code C.
           code D.
```

(b) Duplicate the outermost test on (p)



```
           IF (p)
             PERFORM NESTED-IF
           ELSE
             code C.
           code D.
           .
           .
           .
         NESTED-IF.
           IF (q)
             code A.
           code B.
```

(c) Perform the nested IF statement

Figure 11-5

IFTHENELSE (Con't)

ANS FORTRAN

```
         IF (.NOT.(p)) GO TO 0010
           code A
           GO TO 0020
C        ELSE
 0010      code B
 0020      CONTINUE
C        ENDIF
```

OR

(NO ELSE)

```
         IF (.NOT.(p))GO TO 0010
           code A
 0010      CONTINUE
C        ENDIF
```

Figure 11-6

J3 JOVIAL

```
         IFEITH (p)    $
           BEGIN
             code A    $
           END
         ORIF 1        $
           BEGIN
             code B    $
           END
         END
```

OR

(NO ELSE)

```
         IF (p)        $
           BEGIN
             code A    $
           END
        "ENDIF"
         code B        $
```

Figure 11-7

DO Figures

The DO figures allow iterative execution of a functional block of code (A) based on a logical expression (p). If the test is made prior to the execution of code A, it is a DOWHILE figure. If it is made after execution of code A, it is a DOUNTIL figure.

DOWHILE Primitive

FORMAT

```
DOWHILE        CONDITION-IS-TRUE
        CODE A
ENDDO
        CODE B
```

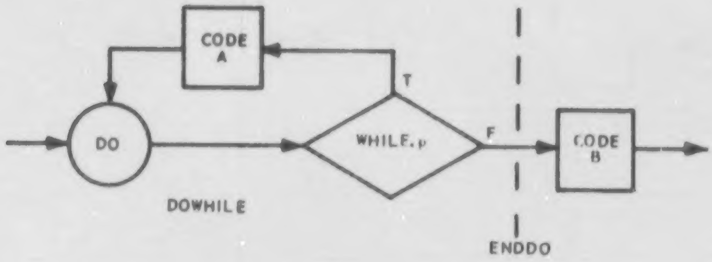


Figure 11-8

When a DOWHILE is executed, the following actions are taken:

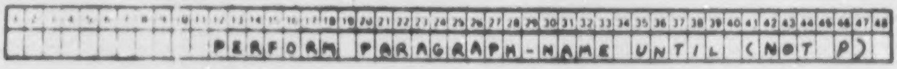
1. As long as the condition is true, code A is executed.
2. When the condition is false, the statement following the ENDDO (code B) is executed.

Code A in the DOWHILE control structure may be comprised of one or more statements and/or control structure figures.

DOWHILE (Con't)

ANS COBOL

All of the loops which are initiated by the PERFORM are of the structured programming DOWHILE format and consequently no simulation is needed. The statement is written as:

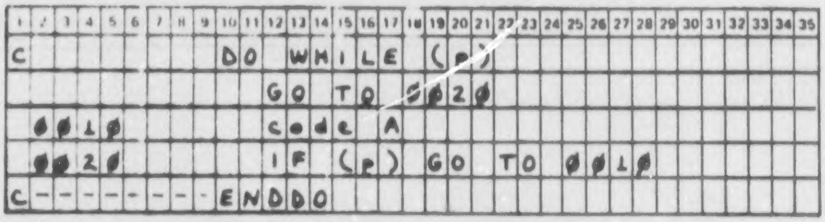


```
           PERFORM PARAGRAPH-NAME UNTIL (NOT P)
```

Figure 11-9

The conditional logic of the DOWHILE requires that the loop be terminated on a "false" condition. (See figure 11-8.) Since the PERFORM. . .UNTIL terminates on a "true" condition, "NOT" logic is used in the UNTIL condition.

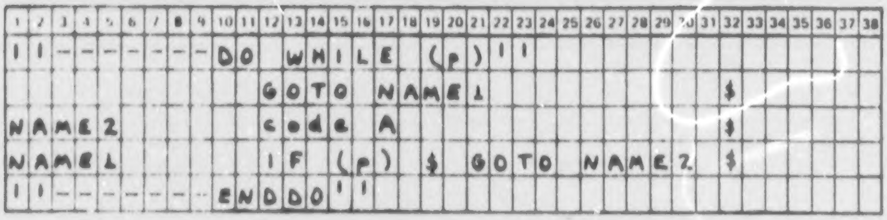
ANS FORTRAN



```
C        DO WHILE (p)
           GO TO 0020
 0010      code A
 0020      IF (p) GO TO 0010
C--------ENDDO
```

Figure 11-10

J3 JOVIAL



```
''-------DO WHILE (p)''
           GOTO NAME1          $
NAME2      code A              $
NAME1      IF (p) $ GOTO NAME2 $
''-------ENDDO''
```

Figure 11-11

DOUNTIL

FORMAT

```
DOUNTIL        CONDITION-IS-TRUE

      CODE A


ENDDO
```

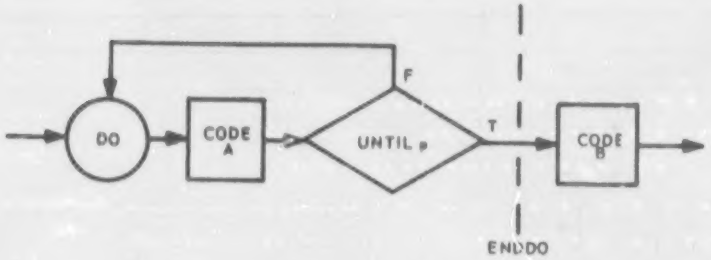


Figure 11-12

When a DOUNTIL is executed, the following action is taken after code A is executed.

1. As long as the condition is false, code A will be executed.

2. When the condition is true, the statement following the ENDDO (code B) will be executed.

Code A in the DOUNTIL control structure figure may include one or more statements and/or control structure figures.

DOUNTIL (Con't)

ANS COBOL

Since all PERFORMs are of the DOWHILE control logic, the DOUNTIL must be simulated. The recommended format is either:

```
           PERFORM PARAGRAPH-NAME.
           PERFORM PARAGRAPH-NAME UNTIL (p).
```

OR

```
           TURN OFF LOOP-SWITCH.
           PERFORM PARAGRAPH-NAME
               UNTIL LOOP-SWITCH IS ON.
```

Figure 11-13

ANS FORTRAN

```
C--------DO UNTIL (p)
 0010      code A
           IF (.NOT.(p)) GO TO 0010
C--------ENDDO
```

Figure 11-14

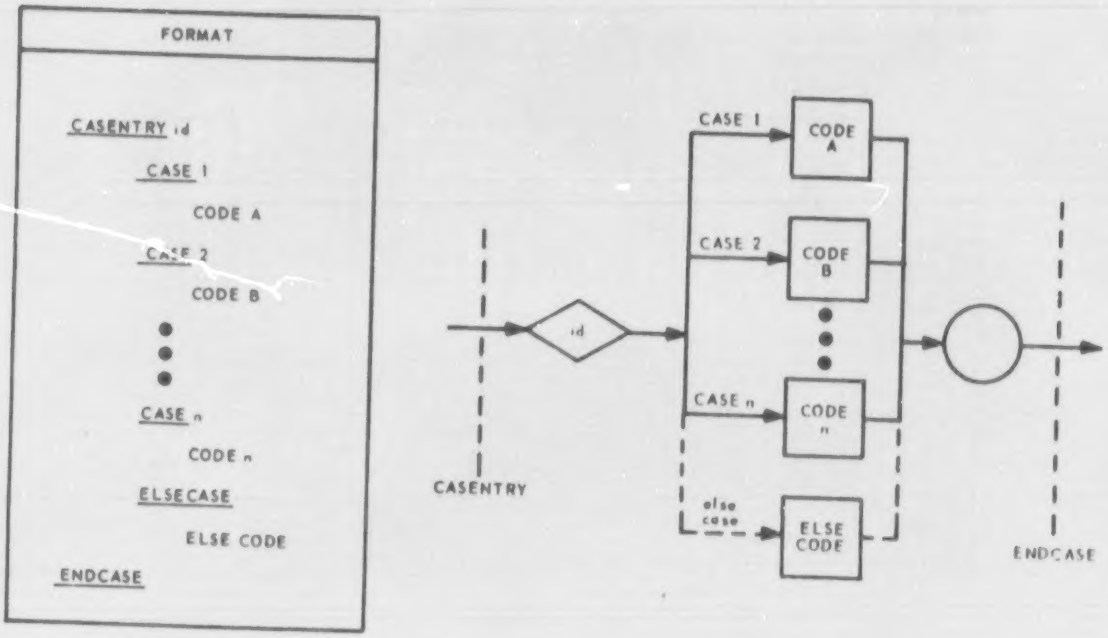
J3 JOVIAL

```
''-------DO UNTIL (p)''
NAME1      code A                  $
           IF NOT (p) $ GOTO NAME1 $
''-------ENDDO''
```

Figure 11-15

CASE or CASENTRY Figure

The CASE structure is a multibranch, multijoin control structure used to express the processing of one of many possible cases.



```
FORMAT

CASENTRY id
    CASE 1
        CODE A
    CASE 2
        CODE B
      •
      •
      •
    CASE n
        CODE n
    ELSECASE
        ELSE CODE
ENDCASE
```

Figure 11-16

When the CASE is executed, control is passed to one of a set of functional modules (code A, B...n) based on the value of an integer variable. If the variable contains a value of 1, control is passed to code A; a value of 2 causes control to be passed to code B. If the value of the variable is not equal to one of the case numbers, control is passed to the ELSE code, if present. Otherwise control is transferred to the statement following the ENDCASE.

CASE (Con't)

ANS COBOL

```
    CASE-PARAGRAPH.
        GO TO CASE-1 CASE-2 ... DEPENDING ON I.
            default code.
            GO TO CASE-PARAGRAPH-END.
     CASE-1.
            code A.
            GO TO CASE-PARAGRAPH-END.
     .
     .
     .
     .
     CASE-n.
            code Z.
    CASE-PARAGRAPH-END.
        EXIT.
```

The above code must be placed out-of-line and is invoked by an in-line PERFORM statement as shown below:

```
 PERFORM CASE-PARAGRAPH THRU CASE-PARAGRAPH-END.
```

Figure 11-17

ANS FORTRAN

```
C---------CASENTRY
            GO TO (0010,0040,0010,0020,0030),I
 0001           default code
                GO TO 0040
C-----------CASES 1 AND 3
 0010           code for cases 1 and 3
                GO TO 0040
C           CASE 4
 0020           code for case 4
                GO TO 0040
C           CASE 5
 0030           code for case 5
 0040       CONTINUE
C--------ENDCASE
```

Figure 11-18

CASE (Con't)

J3 JOVIAL

```
\\---- CASENTRY -----------//
\\------CASE 1             //
         IFEITH (P)        $
           BEGIN
             code A        $
           END
\\      CASE 2             //
         ORIF (q)          $
           BEGIN
             code B        $
           END
         .
         .
         .
\\------ELSECASE           //
         ORIF 1            |
           BEGIN
             default code  $
           END
         END
\\-----ENDCASE-------------//
```

Figure 11-19

NOTICE Each of the control logic primitives has only one entry and one exist from each of the logic structures.

Sequence

The Sequence primitive is nothing more than one statement (which does not break from the program flow) followed by another. Examples include arithmetic statements, input/output statements (without error contingency branches), assignment statements, logical statements, etc.

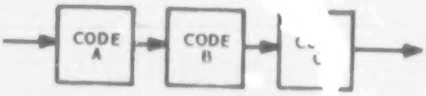


Figure 11-20

482

PRECOMPILERS

In languages which do not contain basic structured programming control figures, two options exist for writing structured code. One is to simulate them with existing facilities of the language as previously discussed, while the second is to use a program which can translate figures written in a structured source code format into an output format which is compatible with the target language assembler or compiler. One type of program which can perform this function is a precompiler which is invoked prior to the compiler and "precompiles" those extensions of the source language which support structured programming into acceptable source language code. Thus, the purpose of a structured programming precompiler is to allow a set of structured programming figures to be used in a language that does not support the figures. This is accomplished by producing, as precompiler output, simulations of the figures in the source language which are then used as input to the language compiler.

The primary purpose of the precompiler is to provide the programmer with the capability to write structured code while requiring as little change in the source language as practical, and to do so in a consistent way in several languages. It is not a function of the precompiler to limit coding by preventing the use of language statements which violate structured programming conventions. Thus, monitoring of the use of explicit branches and other top-down structured programming deviations are not intended to be detected by a precompiler. In addition, anv ambiguities between the source language and structured programming statements are presumed to have been removed from the precompiler input. Thus an IF statement is presumed to have a corresponding ENDIF in all such source code.

The precompiler's basic function is to support the use of four standard structured programming figures.

1. IFTHENELSE
2. DOWHILE
3. DOUNTIL
4. CASE

The relationship between the precompiler and its inputs and its inputs and outputs is graphically displayed by the following:

Precompiler Input

A precompiler accepts the code needed to implement the primitives described earlier, plus all statements native to the language being used. Error detection by the precompiler is limited to the precompiler code. Any errors in the native language statements are detected by the compiler or assembler. The programmer must insure that all verbs common to both the native language and the precompiler code are in the precompiler format - i.e., the IF statement. The precompiler adds the capability of structured code to an existing language without changing the features of that language.

Should it be desired to mix both structured and unstructured code, this may be done by placing the unstructured code in the library and then bringing it into the program by means of the include function. The INCLUDE statement which may be processed by either the library system or the precompiler, will have, in addition to the name of the code segment which is to be placed in-line, an additional indication that the code to be processed is unstructured. The implication of the above is that the top most segment in the tree structure is presumed to be in the required structured program format and from then on, the format of the nested modules is identified by the programmer. The precise formats for these indicators will be outlined in "Precompiler Specifications," Volume II of the Structured Programming Series which covers the various languages in Section 4.

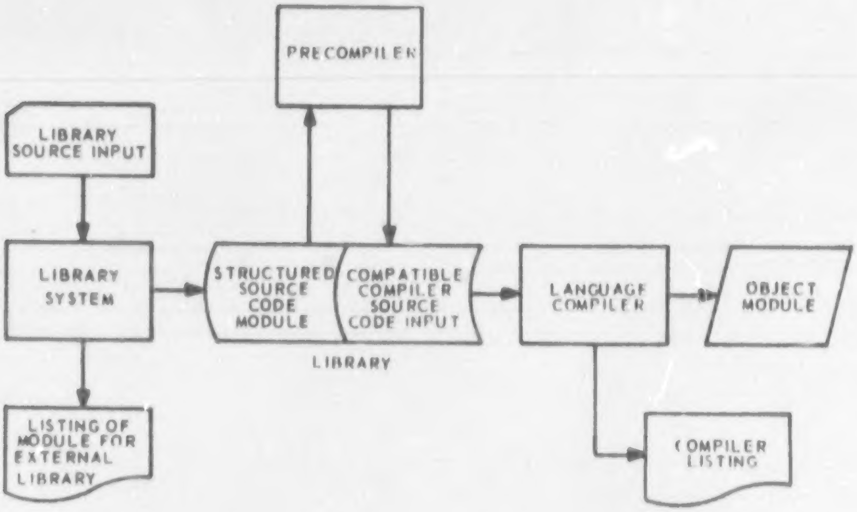

PRECOMPILER
LIBRARY SOURCE INPUT
LIBRARY SYSTEM
STRUCTURED SOURCE CODE MODULE
COMPATIBLE COMPILER SOURCE CODE INPUT
LIBRARY
LANGUAGE COMPILER
OBJECT MODULE
LISTING OF MODULE FOR EXTERNAL LIBRARY
COMPILER LISTING

Figure 11-21

Indentation

Strict attention should be paid to the indentation of the logic structures on the printed page so that logical relationships in the coding correspond to physical position on the listing, and a pictorial representation of the logic can be gained from the indentation.

Automatic indentation is not a function of the precompiler, but is a desirable feature of a library system. This feature eases the burden of modifying code since alignment is handled automatically when a source module listing is produced. It also corrects any manually produced erroneous indentation and standardizes listing formats. However, when new modules are being produced for the first time, the programmer should continue to write indented source code as an aid in the coding process.

Precompiler Output

Compiler Input - The precompiler produces and places in a library file (when the facility exists) an output source module to be used as input to the appropriate compiler. All standard program figure usage will have been resolved into valid source language statements as part of the precompiler's output. The programmer-written source code listing representing input to the precompiler should be used for debugging purposes instead of the compiler output listing.

Simulated Structured Code - The precompiler output will, where practical, be simulated structured code following the conventions specified in the RADC Structured Programming Language Standards section (Section 4). This assists the programmer in relating diagnostic messages on the compiler listing to the corresponding module in the precompiler source input. In addition, if it becomes necessary to use a compiler source listing, the code layout closely resembles the programmer's own source program.

Language Considerations

This subsection discusses language considerations when a precompiler implementation approach is used for a higher level language.

JOVIAL J3

It is suggested that the precompiler IFTHENELSE standard program figure be used in place of the combination of an IFEITH and ORIF conditional statement, and the IF conditional statement where the ELSE clause is omitted. Although the IFEITH/ORIF combination (multiple ORIFs are permitted) provides more capability than the IFTHENELSE standard program figure, an attempt is being made to provide a near uniform structuring facility via a precompiler. In fact, a precompiler would translate the IFTHENELSE into an IFEITH/ORIF combination. Another reason for the selection and use of the structured program figure is the addition of the ENDIF delimiter which adds to the readability.

As with FORTRAN, JOVIAL J3 does not provide a DOWHILE or DOUNTIL capability; i.e., repeated execution of a series of statements with a leading loop test (DOWHILE) or a trailing loop test (DOUNTIL). JOVIAL J3 only provides an indexed looping capbility. Thus, the DOWHILE and DOUNTIL figures should be used where appropriate.

The CASE standard figure should be used in place of the in-line JOVIAL J3 simulation of the CASE presented in figure 11-19.

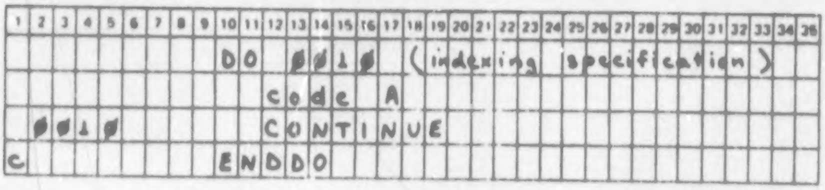
In general, any JOVIAL J3 statement which violates structured programming concepts should be avoided. For the most part, an attempt should be made to avoid branching. For a discussion of recommended JOVIAL J3 coding conventions, refer to the section on coding conventions.

ANS FORTRAN

The precompiler IFTHENELSE standard program figure should be used in place of the FORTRAN logical IF and arithmetic IF statements. It provides more capability than the FORTRAN logical IF because of its ELSE clause. The IFTHENELSE figure, with nesting sometimes required, will also provide more capability than the FORTRAN arithmetic IF and allows a clearer statement of the condition tested.

The capabilities of the DOWHILE and DOUNTIL, i.e., repeated execution of a series of statements with a leading loop test (DOWHILE) or a trailing loop test (DOUNTIL) are not provided by the FORTRAN DO. The FORTRAN DO, a specialized DOUNTIL, only provides iteration based on the range of an index. Thus, the DOWHILE and DOUNTIL should be used where appropriate. The FORTRAN DO may, however, be used for its specialized indexing capability. The code structure previously recommended when using the FORTRAN DO statement would be:

485



```
         DO 0010 (indexing specification)
           code A
 0010      CONTINUE
C        ENDDO
```

Figure 11-22

The CASE standard program figure would be used in place of the FORTRAN simulation of the CASE figure using a computed GOTO statement and GOTO statements.

In general, any FORTRAN statement which violates structured programming concepts should be avoided. For the most part, an attempt should be made to avoid unconditional branching as provided by GOTO, computed GOTO, arithmetic IF, ASSIGN and ASSIGNED GOTO, and ENTRY and RETURN i. The use of programmer generated statement numbers and null statements (CONTINUE) should be avoided wherever possible. For a complete discussion of recommended FORTRAN conventions, refer to the section on Coding Conventions.

ANS COBOL

The precompiler IFTHENELSE standard program figure should be used in place of the COBOL IF statement since the COBOL IF statement does not provide a proper nesting capability (refer to the nested IFTHENELSE discussion in the previous section).

The COBOL PERFORM statement with the UNTIL option, where the looping condition is tested prior to execution of the code within the loop, provides a capability similar to that of the DOWHILE standard program figure. The difference lies in the fact that the test condition is inverted between the two and the PERFORM passes control to an out-of-line procedure. Inverting the condition along with use of the word UNTIL would tend to obscure the intent of the test condition. Thus, the DOWHILE standard program figure should be used.

Using a PERFORM statement followed by another PERFORM statement with the UNTIL option, when both refer to the same procedure name, provides the same capability as the DOUNTIL standard program figure. The DOUNTIL standard program figure should, however, be used for sake of clarity.

The CASE standard program figure, which is not part of ANS COBOL, should be used in place of the COBOL simulation which is accomplished by using a GOTO statement with the DEPENDING ON option and GOTO statements.

Statements which violate structured programming concepts should be avoided. For the most part, an attempt should be made to preclude branching. For a discussion of recommended COBOL coding conventions, refer to the section on Coding Conventions.

479

486

CODING CONVENTIONS

Additional Recommended ANS COBOL
Coding Conventions

Restricted ANS COBOL Statement Usage

In order to preserve the concept of structured programming, it is recommended that the general usage of those statements in COBOL which permit changes of sequential control be restricted to an exception basis only. Thus, in addition to the GOTO, the ALTER statement should also be limited in its usage.

Program Organization

The structure of a COBOL program is such that many of the rules for program organization have been predefined. For instance, all data must be specified in the DATA DIVISION. Furthermore, within this section, the formal rules which define the permissible hierarchical data structures are sufficient to preserve the readability requirements of structured programming. However, within the PROCEDURE DIVISION (with the exception of the DECLARATIVE SECTION), the rules of COBOL permit the ordering of the PERFORMed code blocks to be completely flexible.

If the program is being developed with the aid of a library system, the order in this division is less critical since all that appears after the top most segment are COPY statements. The functions which exist in the copied code and the functions which are nested within them are determined by examining the small code segments which are present as printed listings of members in the source code library rather than on the compiler output listing even though it is still true that the resolution of the COPY statements by the compiler will produce a complete source program as one of the compiler outputs.

However, for a development process in which no random access library exists, the ordering of the segments of PERFORMed COBOL paragraphs in the procedure division is more critical. This is because the source listing under this condition is a single sequential data set. At present, the suggested sequence is initially by nested level for 2 or 3 levels (depending on the program's complexity) and alphabetically thereafter.

PERFORMed paragraphs should be separated from the main body of code, and from other PERFORMed paragraphs, by at least two blank lines. Logically noncontiguous paragraphs (other than those used in the CASE figure) should be separated by at least one blank line.

Comments

One of the primary intents of the developers of the COBOL language was to produce a self-documenting language. When this is coupled with the discipline of structured programming the resulting programs should be even more readable. Therefore, it is recommended that the use of comments in the form of NOTE sentences and NOTE paragraphs be held to a minimum. When they are used, they should not interfere with the readability of the program itself. This may be done by using blank lines to insure that the NOTE text stands apart from the program proper and by indenting the textual commentary to Column 35 or 40.

487

Indentation and Formatting Conventions

Variables and structures defined in the DATA DIVISION should be arranged in a meaningful order. This order could be alphabetic, by class such as the days of the week, or any other class format. A suggested set of indentation rules for data items is as follows:

1. General Format. All level 77 and 01 variables should have their level numbers in columns 10-11 and names starting in column 14. The PICTURE clause should be between columns 32-45, depending on the length of the longest variable name. All other clauses used should follow the PICTURE clause with normal spacing. If more than one line is needed for a variable's definition, the second and succeeding lines should be indented from the PICTURE clause as follows:

```
  77  RECORD-COUNT      PICTURE 9(7)V99
                            COMP SYNC.
```

Figure 11-23

2. Structures. Each successively lower level in a structure should be indented four columns from the next higher level. Level numbers should precede each variable name in the structure on the same line and two columns before it as follows:

```
  01  EMPLOYEE-RECORD.
      02  NAME.
          03  FIRST     PICTURE X(10).
          03  MIDDLE    PICTURE X.
          03  LAST      PICTURE X(20).
      02  ADDRESS.
          03  STREET    PICTURE X(15).
```

Figure 11-24

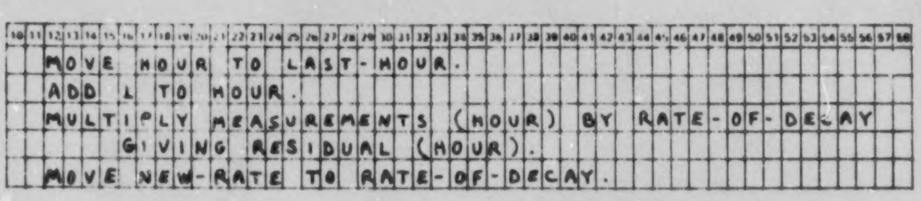
When condition-names (level 88 items) are used, they should be indented and written with single spacing between words.

```
          03  TYPE-CODE   PICTURE X.
            88  NEW-TYPE VALUE 'B'.
            88  OLD-TYPE VALUE '2'.
```

Figure 11-25

481

The indentation recommendations for the structured programming figures which appear in the PROCEDURE DIVISION were previously specified in the section where they were described. When programming the code blocks which represent the SEQUENCE in structured programming, each COBOL statement should start a new line. Any statement not subject to indentation rules starts in the same column in which the statement above it started. Only one statement per line is permitted. If any statement must be continued on the next line, the continued portion should be indented 4 to 6 columns so as not to obscure the COBOL verb as follows.



```
MOVE HOUR TO LAST-HOUR.
ADD 1 TO HOUR.
MULTIPLY MEASUREMENTS (HOUR) BY RATE-OF-DECAY
    GIVING RESIDUAL (HOUR).
MOVE NEW-RATE TO RATE-OF-DECAY.
```

Figure 11-26

Additional Recommended ANS FORTRAN Coding Conventions

Restricted FORTRAN Statement Usage

In order to maintain structured programming concepts, it is recommended that certain allowable ANS FORTRAN statements generally not be used except as required in the previous definition of the standard program figures and summarized below. For the most part, an attempt is made to preclude unconditional branching not necessitated by standard program figure definition.

The GOTO statement is used in the definition of the following standard program figures: IFTHENELSE, DOWHILE, DOUNTIL, and CASE. The computed GOTO statement is used in the definition of the CASE standard program figure. It should be an objective not to use these statements except in those figures.

The ASSIGN and ASSIGNED GOTO statements provide an unconditional branching capability. The arithmetic IF statement is not necessary because the IFTHENELSE standard program figure, with nesting sometimes required, will provide the same capability. Use of these FORTRAN statements should be avoided.

The recommended use of the DO statement as a specialized DOUNTIL is covered in a previous subsection. Other usage of the DO is not recommended.

The CONTINUE statement is used in the definition of the IFTHENELSE and CASE standard program figures. In addition, it is sometimes required by a DO (specialized DOUNTIL) statement. No other use of the CONTINUE should be necessary.

Program Organization

These conventions provide for the organization of a FORTRAN source program into a set of segments for compilation. Any FORTRAN program requires a certain ordering of the statements within the program. A suggested further restriction to that ordering for the sake of readability, clarity, and consistency appears below.

1. If this is a subprogram, the first card must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

2. Any COMMON statements, each followed by all type, DOUBLE PRECISION, and EQUIVALENCE statements related to it follow. No dimension information is to appear on a COMMON statement. The COMMON statement will be used only to declare the order of arrays and variables within the COMMON. Blank COMMON is to be declared first, followed by all labeled COMMONs in alphabetical order.

Any explicit specification (type) statements and DOUBLE PRECISION statements will be arranged in alphabetical order of the variables or arrays within each of the types. They will be defined in the following order: COMPLEX, DOUBLE PRECISION, REAL, INTEGER, and LOGICAL. All dimensioning information should be included on the type or DOUBLE PRECISION cards. All variables or arrays should be explicitly declared, and the DIMENSION statement should not be used in place of a type statement.

Following each type or DOUBLE PRECISION statement, any EQUIVALENCE statements required for that type statement are included. A blank comment card should be used before and after the EQUIVALENCE statements to set them off from the surrounding definitions.

3. Once all COMMON declarations are made, the program local declarations are made using the same conventions.

4. Following all program local declarations, all EXTERNAL declarations will be made.

5. Any DATA statements for program local arrays and variables follow.

6. All FORMAT statements follow.

7. Any statement function definitions come next and complete the nonexecutable code.

8. Segments containing executable code follow in order. The last segment must contain an END card.

9. If desired, subprograms may follow as part of a multiple compilation. The organization of each subprogram should follow the rules given above.

Comments

Comments should be used to enhance the readability and understanding of a program (e.g., to define variables or their special settings). In general, when they are used, they should be grouped together as a prologue to the code segment. If they must be interspersed within the code, they should be inserted as a block which begins in a column near the middle of the page (e.g., column 35 or 40) so as not to interfere with the indentation and readability of the program proper which may be scanned near the left margin. Blank comment cards should be used when they enhance readability.

Statement Numbering

As much as possible, statement numbers are to proceed from lowest to highest as a program is read. It is recommended that statement numbers be four digits long, be placed in columns 2-5, and be incremented by 10 rather than be consecutive.

Continuation Cards

ANS FORTRAN permits up to 19 successive continuation cards per statement. The continuation column should be used to indicate the order of the cards. This may be done by placing a numeric character in column 6 (the continuation indication column) in ascending sequence (i.e., 1-9) and if additional characters are necessary, using in order the alphabetic characters A-J.

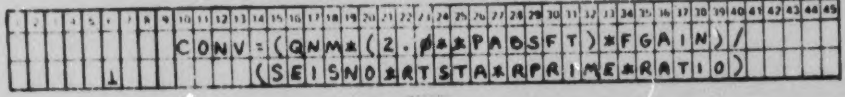
The body of the continuation card should be coded so as to enhance the readability of the program unit. In the following subsections are some suggestions in regard to special cards, but, in general, no continuation card should contain information to the left of the statement identifier on the first card.

Statements

Each statement should begin in an even-numbered column. Nonexecutable statements should begin in column 8. If the program is a subprogram, the FUNCTION, SUBROUTINE, or BLOCK DATA statement should begin in column 8, and the corresponding END statement should also begin in column 8; in these cases, the first executable statement will begin in column 10. Otherwise, the first executable statement of a program should begin in column 8, as should the END card. Succeeding executable statements are indented according to the coding examples in the standard program figures section and the rules given below.

Assignment Statements

If a statement is continued, the second and following lines should be indented by six columns. For example:



```
         CONV=(QNM*(2.0**PADSFT)*FGAIN)/
     1         (SEISNO*RTSTA*RPRIME*RATIO)
```

Figure 11-27

In a continued assignment statement, the first card (and each card which will be followed by another continuation card) should have an operator as the last character. As much as possible, all parenthetical expressions should be on a single card. In no case should a left parenthesis be the last character on a card which is to be continued.

COMMON Statements

The COMMON card will begin in column 8 with the identifier followed by a blank in column 14 and a "/" in column 15. The next six columns are reserved for the label which will be left justified in this field. Column 22 will contain another "/" and column 23 will contain a blank. If blank COMMON is desired, code the slashes but leave the label field blank. Columns 24 through 50 will contain the names separated by commas. Commas will appear in columns 30, 37, and 44. If a continuation card is necessary, a comma will appear in column 23 on that card. The names will be left justified within each field.

Type Statements

Dimension information for each array is to be contained on a single type card. A card will contain one and only one name. A type statement may be continued, thus allowing 20 names to be declared for each statement.

The type statement will be coded beginning in column 8. The name should begin in column 16. If it is a continuation card, a comma should be inserted into column 15. When using a DOUBLE PRECISION statement, the name should begin in column 25. If a continuation card is required, a comma should be inserted into column 24.

FORMAT Statements

The FORMAT identifier will be coded beginning in column 8. The first parenthesis will appear in column 16, and the format information itself begins in column 18. As much as possible, a position code and its associated format code should stand alone on a card. Continuation cards should be used liberally. For example:

```
 9060  FORMAT  ( 1H0,20X,8HMONTH = ,I2,
     1                4X,6HDAY = ,  I2,
     2                4X,7HYEAR = , I4)
```

Figure 11-28

READ or WRITE Statements

In all cases a READ or WRITE statement should have its subject data set reference number (DSRN) contained in a variable. The use of hard coded DSRNs is discouraged from the standpoint of visibility and parameterized coding. The DSRN variables may be loaded with a DATA statement.

The list portion of the READ or WRITE statement must be expressed in as simple terms as possible. Liberal use of blanks and continuation cards is encouraged in order to increase readability.

IF Statements

Multiple conditions in the predicate of an IF statement should occur on separate cards, with an operator occurring as the final item in each card to be continued. For example:

```
         IF (A.GT.B .OR.
     1       C.LT.D .AND.
     2       X.EQ.J
     3      ) GO TO 0100
```

Figure 11-29

Notice also that the GOTO statement follows the closing parenthesis.

DATA Statements

Only one variable may be specified on a DATA statement card. The DATA identifier will be coded beginning in column 8. The variable name will be coded beginning in column 14. The slash indicating the beginning of the data will be coded in column 20. For example:

```
       DATA  YEAR  /1975/
```

Figure 11-30

Additional Recommended JOVIAL J3 Coding Conventions

Restricted JOVIAL J3 Statement Usage

In order to maintain the concept of structured programming, it is recommended that certain statements which are part of the JOVIAL J3 language be used on an exception basis only. These statements all involve the language capability which effects change of control whether on an explicit or implicit basis. Thus, in the JOVIAL J3 language the usage of the GOTO, TEST, and special compound statements should be limited as much as possible to the simulation of the DO and CASE figures in the manner indicated in the previous section. The limitations placed on the special compound statement also result in a limited usage of the incomplete FOR statement. This is also in agreement with the discussion of the DO figure simulations. Finally, the use of SWITCH declarations to invoke SWITCH declarations is also a transfer of control mechanism and should be minimized.

Program Organization

The data and executable statements which comprise a program should be in a meaningful order and in a format that can be easily understood by anyone who wishes to read and code. More important than a single set of rigorous rules covering all projects is a consistent set of conventions for a single project. The following is a suggested ordering of the various components which comprise a JOVIAL J3 program.

1. The START and TERM brackets (if they appear in a segment) should start in column 1. Except for these, each code segment should start to the right of column 1 in a consistent manner. Indentation for figures within a code segment should be cumulative within the segment.

2. The suggested sequence of program segments which follows is based upon the assumption that no INCLUDE capability exists.

   a. START bracket.

   b. All data declarations (ITEM, ARRAY, TABLE, SWITCH, etc.) used in the first segment or by more than one lower level segement. The data declarations should be arranged in a meaningful order, including table items in an ordinary table. This is not necessarily true for defined table items or for table items in an input or output buffer where the order of the items is defined by the arrangement of data within the record.

   c. Main line code.

   d. Lower level segments. The segments (procedures, functions, CLOSE declarations) should be arranged in a meaningful order by level, with the lowest level segments appearing last. All data declarations within a segment should appear first within a segment and follow the same rules as the data declarations in the main segment.

   e. TERM bracket (and FINIS or STOP statement).

Comments

Comments should be used where necessary to enhance the readability and understanding of a program. In general, when they are used they should be grouped together as a prologue to the code segment. If they must be interspersed within the code, they should be inserted as a block which begins in a column near the middle of the page (e.g., column 35 or 40) so as not to interfere with the indentation and readability of the program which may be scanned near the left margin.

Indentation and Formatting Conventions

As with the program organization recommendations, the formatting guidelines should also be flexible enough to be easily adapted to any project. The data declaration rules which follow should be taken as an example of how a given project may wish to achieve consistency in formatting.

1. Tables and arrays should be in a meaningful order by table/array name.

2. No single items should be declared except parameter items. Temporary working storage items should be declared in one or more compiler-allocated tables.

3. Items within a programmer-allocated table may be in alphabetical order. Items within a compiler-allocated table must be in alphabetical order.

4. Preset values for items should occupy one or more additional lines following the item declarations.

5. Table names should have two character names. All items within a table should start with those same two characters.

6. Internal tables (noncompool) should be named by a letter-numeric.

7. "Like" internal tables duplicating a compool table should be named letter-letter-numeric, where the first two characters define the compool table name and the numeric indicates an internal. Equated internal tables used to define temporary compool table items should also use this same naming scheme.

8. Item names should normally be five characters in length.

The indentation of the code comprising the simulated structured programming figures should be as indicated in the previous section. In general, any statement not subject to the indicated indentation formats starts in the same column as the statement above it. Only one statement per line is permitted except where used in the certain of the control logic structure simulations. The continued portions of statements requiring more than one line are all indented in a block beneath the first line of that statement. This is particularly true of the conditionals which follow the IFEITH, ORIF, and IF primitives to clarify the range of statements delimited by these primitives.

POSSIBLE STRUCTURED CODING PROBLEMS

Like top-down implementation and program design languages, structured coding can also have its problems. In this subsection, some of the problems and possible solutions are discussed.

One problem that has been reported by many programmers is that "it is difficult to implement" a program design in structured coding. It can be a real problem especially if a programmer has been trained in a traditional manner of coding. The solution lies within the programmer. There will be a learning curve involved in using structured coding. And, the programmer must want to learn. It will take time and practice.

Another problem is programmer ego. There is a certain amount of pride of authorship in just about every programmer. However, much emphasis has been placed on the "tricky code." This can hurt the maintenance programmer in the end. Pride can be realized from writing code that is easily maintained and understood both by the author and his fellow workers and successors. Thus, the old "ego" is still saved.

Closely related to the ego problem is the resistance to change. (Sound familiar?) For obvious reasons, some programmers do not want to change their programming habits -- even when some are bad habits! Excuses include: the "if it was good enough for dad, it is good enough for me" attitude, laziness, ego, the "I don't understand it and therefore it can't be any good" attitude, and the list goes on. Solution? It will depend largely on the programmer who has this problem. Is he (or she) willing to give a "new" thing a trial period? Is he willing to look at structured code (or anything for that matter) with complete objectiveness? Only the programmer can answer these questions.

Inefficient machine usage has also been said to be a problem; i.e., structured code requires more core and time, and indeed this could very well be true. To see what can be done about it, let's look at two things. First, look at the overall time efficiency. A valid question that might be asked "is saving 1 millisecond of execution time worth five or more hours additional programming time, plus testing and debugging time?" The answer will depend upon program usage and other factors. The second thing to look at is core efficiency. This solution (?) is discussed later in this chapter.

One of the best ways to make a mountain out of a molehill is to develop the structure for a program using only structured coding. Don't do it! First, the structures are variable in form and quality (due to a lack of guidelines and discipline). Second, the structures are not available in advance; the structure emerges from the code. Third, the structures tend to be input-driven rather than transform-centered, transaction-centered, pathologically connected, and sequential in nature (as opposed to being functional). Solution: Develop the structure first using design aid tools -- i.e., HIPOs, structure charts, data flow graphs, etc. Then, use structured code to implement the design. If a program design language (see PDLs in chapter 10) is used prior to using structured code, the coding becomes an easy process since the same primitives are used in PDL and structured code. Last, develop guidelines for implementing the control logic figures and stick with the guidelines. Don't try to take short cuts with "tricky code."

CODING EFFICIENCY CONSIDERATIONS

INTRODUCTION

One of the most frequent complaints against structured programming is that it results in inefficient code. When programmers speak of such inefficiencies, they are referring to the executable code which either runs too slowly (has an excessive path length) or occupies too much core, or both. Since the majority of the languages addressed in this report are high level ones, the efficiency questions which are addressed in this section should be evaluated with this fact in mind. It also should be recalled that structured programming is only another way of writing branch instructions and therefore the claims of inefficiency must address the basic problem of conditional branching and not the more voluminous statements which perform the basic sequence block processing.

It is obvious that if a compiler generates inefficient code, then regardless of whether or not the source program is structured, the result is still inefficient code. Therefore, the question of efficiency in higher level languages is one of whether structuring has added an excessive amount of inefficiency. This question is addressed from the following four points of view.

1. Compiler code generation.

2. Simulation of structure figures and top-down requirements.

3. Improper use of the source language.

4. The virtual storage environment.

Compiler Code Generation Inefficiencies

The problem of inefficiencies which arise from a failure to take advantage of the hardware architecture is one which cannot be controlled for higher level languages since this is a function of the compiler code generation and is unaffected by structuring. For instance, consider the CASE control logic structure. If the target computer has an indexed branch capability and the compiler has a CASE statement, the compiler could take advantage of this fact in its code generation. This is evident when comparing the following code:

```
CASENTRY (Register holding case number)
  CASE 1
    Code A
  CASE 2
    Code B
     .
     .
     .
  CASE n
    Code n
ENDCASE
```

and the compiler generated code:

```
                SHIFT to adjust number to computer word length
                BRANCH TO LABEL1 INDEXED by shifted number
     LABEL3:      Code A
                BRANCH LABEL2
     LABEL4:      Code B
                BRANCH LABEL2
                  .
                  .
                  .
     LABELn:      Code n
     LABEL1:    BRANCH LABEL2
                BRANCH LABEL3
                BRANCH LABEL4
                  .
                  .
                  .
                BRANCH LABELn
     LABEL2:    exit  point
```

From the generated code, it can be seen that to select the correct CASE number requires that the number first be adjusted in order to make it compatible with the target computer word length (probably a SHIFT instruction). This is followed by an indexed BRANCH to the correct unconditional BRANCH in a branch vector list (a sequence of branch instructions), a total of three instructions.

If the language does not contain a specific CASE verb, then this particular architectural feature of the target computer cannot be used and a corresponding loss of efficiency results in simulating this control logic structure. For instance, if the language is FORTRAN and the CASE is simulated by the FORTRAN computed GOTO statement, the code efficiency is a function of what the FORTRAN compiler generates for this verb and not structured programming. In the extreme, the CASE can be simulated by a series of IFTHENELSEs such as with the JOVIAL J3 IFEITH statement (one ORIF for each case number which may be present). This implementation would probably be the most inefficient since code must be present to test for each possible CASE value.

Another example which illustrates that efficiency in high level languages is dependent upon the compiler can be shown using COBOL as an example. The looping capability in this language is invoked with a PERFORM verb. If a precompiler were used, the DOUNTIL loop:

```
         DO UNTIL (p)
           code A.
         ENDDO
         code B.
```

would result in the following generated code:

```
       PARAGRAPH-NAME-1.
            code A.
            IF NOT (p) GO TO PARAGRAPH-NAME-1.
          Code B.
```

Figure 11-31

If this control logic is simulated by the code shown earlier in this chapter, the question which should be addressed is whether the efficiency of the loop is a function of what the COBOL compiler generates for a PERFORM versus what it generates for an IF and GOTO statement.

From the examples in this section, it is evident that efficiency is dependent upon the code generated by the compiler. Not only can efficiency vary from one machine to another, but different versions of a compiler for the same computer can also affect the efficiency.

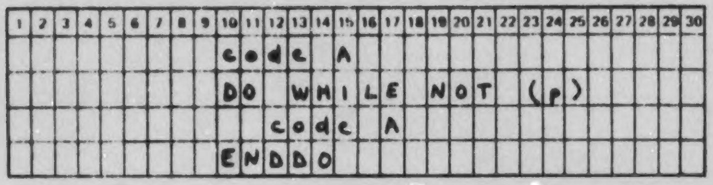
Simulation Inefficiencies

A second source of inefficient code arises when it is necessary to simulate a structuring or top-down capability because it does not exist in the source language. For example simulating a DOUNTIL:

```
         DO UNTIL (p)
           code A
         ENDDO
```

Figure 11-32

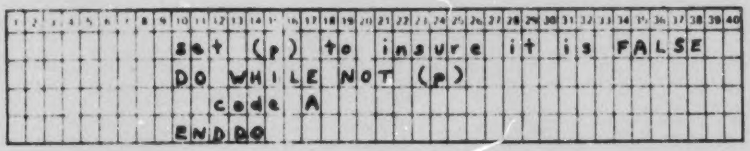
in a language which only has a DOWHILE. The DOUNTIL control structure requires that code A be executed at least once. This can be insured in at least one way and possibly two. Both simulations use a DOWHILE and negate the condition. The first method is to duplicate code A before a DOWHILE statement:



```
code A
DO WHILE NOT (p)
  code A
ENDDO
```

Figure 11-33

The second simulation (which may not always be possible) is to insure that condition (p) is false so that code A is always executed at least once as follows:



```
set (p) to insure it is FALSE
DO WHILE NOT (p)
  code A
ENDDO
```

Figure 11-34

In these simulations there is at least one and possibly two sources of inefficiency. The first involves the negation of the condition (p) and, as with the examples in this section, is dependent upon how the compiler evaluates the looping condition to determine whether it is true or false. This can be illustrated by the following simple example. If the conditional statement is of the form p AND q, and if a test determines that p is FALSE, then the entire expression is FALSE and it is not necessary to execute the code which tests q. However, if this expression is negated by placing it inside of parentheses and preceding it with NOT as follows:

NOT (p AND q)

then it becomes necessary to evaluate both p and q before testing the truth of its negative. In this case, more code must be executed. However, if the compiler is sophisticated enough to manipulate the negated expression, it can still generate code such that there is no increase in the amount of code executed when the expression is negated and no corresponding efficiency loss in execution. The second source of inefficiency, however, has nothing to do with the code generated by the compiler. It is simply that in the first simulation, code A must either be duplicated, in which case more core space is required, or else code A is made a subroutine in which case space and execution time are required for the linkages. In the second simulation, condition (p) must be set to FALSE which involves both space and execution time.

Improper Source Language Usage

The example shown in this section is an illustration of poor programming and as a consequence can occur whether the code is structured or not. However, it has been included in order to show how much easier it is to detect poor code when it is structured. Consider the following code:

```
         BRANCH (p) FALSE to LABEL1
         CALL code A
         BRANCH to LABEL2
LABEL1:  BRANCH (q) FALSE to LABEL2
         CALL code A
LABEL2:  entry point
```

and its structured equivalent:

```
         IF (p)
           CALL code A
         ELSE
           IF (q)
             CALL code A
           ENDIF
         ENDIF
```

Figure 11-35

The branching that is generated by a translation of the structured code in the above example is the same as that which is shown for the unstructured code in the example which immediately precedes it. The code is inefficient in both cases but it is not easily spotted in the unstructured version. In the structured version, however, it is very evident.

Thus, the following code is functionally equivalent:

```
         IF (p) OR (q)
           CALL code A
         ENDIF
```

and its translation would be:

```
         BRANCH (p) TRUE to LABEL1
         BRANCH (q) FALSE to LABEL2
LABEL1:  entry point
           CALL code A
LABEL2:  entry point
```

Figure 11-36

A comparison of the translation with the unstructured code shows first that the two calls of code A have been reduced to one and second, the unconditional branch to LABEL2 has disappeared. Furthermore, if this is the only place that code A is called, it is possible to INCLUDE code A in-line rather than calling it and thus save the subroutine linkages as well. The inefficiency obviously arose because of the failure to use the boolean capability of the source language and in this respect, the source language was used improperly.

Virtual Storage

The problems which are encountered using structured programming in a virtual storage environment are no different than those using unstructured code. In both cases, it is necessary to recognize that to minimize paging and "thrashing," it is necessary to control the placement of both data and code, so that the most frequently accessed portions of the program are concentrated in contiguous locations. Since structured programming does not restrict where code may be placed (it may be included in-line or placed out-of-line and invoked as a subroutine), the control of code placement is unaffected.

However, it should be pointed out that the guidelines as to data placement which are specified in the various languages standards may have to be modified in a virtual storage environment. For instance, in the FORTRAN standards, a suggested ordering of a code module in terms of COMMON, NAMELIST, DATA, FORMAT and code statements is specified. In a virtual storage environment, the recommended sequence may not be optional and should therefore be modified. A similar situation exists with COBOL. The flexibility which exists with regard to the placement of data is limited by the COBOL language itself since all data specifications must be restricted to the DATA DIVISION. However, the guidelines given in the previous section with regard to the placement of PERFORMed paragraphs may not be optional in a virtual storage environment. Rather than ordering such paragraphs by level or alphabetically, they are better grouped by expected frequency of execution. (See Chapter 9, Packaging.)

Efficiency Summary

In summary, programming inefficiencies introduced with structured code include those caused by either having to simulate one of the capabilities required because it does not exist in the source language or because the true control logic which the programmer wishes to execute does not exist as a primitive control structure in the language. When these are weighed against the inefficiencies resulting from compiler code generation or poor programming, they are probably negligible in comparison.

As previously stated, structured programming does not guarantee good programming. It is still a function of the programmer. However, because of the structuring, it is far easier to detect and correct problem code.

It has been demonstrated many times, that as a result of program maintenance or enhancements, blocks of code become isolated and cannot be reached from any part of the program. In fact, in order to guard against this, some assemblers and compilers contain a flow trace analysis in order to alert the programmer when this occurs. With structured programming, a path exists to all blocks of code and such traces are not required. Whether all such blocks will be executed depends on the conditional logic specified by the programmer, a factor which is not only beyond the control of structured programming but is also not detectable by a flow trace analysis.

Examples of Structured Coding

Figure 11-37 illustrates the difference between "traditional" COBOL and structured COBOL for the same section of code. Two differences that are immediately evident are the indentation and lack of GOTOs in the structured code. Upon further examination, you will notice the conditions of the IF statements are different.

At first, programming without using GOTO statements takes concentrated effort since it requires a change in your thinking process. With experience, structured coding becomes easier. Indentation of the code will be beneficial for programmers writing structured code.

TRADITIONAL COBOL

```
PARA - 1.
     IF NUMBER > 20; GO TO PARA - 3.
     IF NUMBER > 10; GO TO PARA - 2.
     MOVE A TO B.
     SET IND TO 1.
     GO TO PARA - 4.
PARA -2.
     MOVE A TO C.
     SET IND TO 3.
     GO TO PARA - 4.
PARA - 3.
     MOVE A TO D.
     SET IND TO 5.
PARA - 4.
     .
     .
     .
```

STRUCTURED COBOL

```
PARA - 1.
IF NUMBER ≤ 20;
      IF NUMBER > 10;
           MOVE A TO C
           SET IND TO 3;
      ELSE
           MOVE A TO B
           SET IND TO 1;
ELSE
      MOVE A TO D
      SET IND TO 5.
PARA - 4.
      .
      .
      .
```

Figure 11-37

This next example applies TDSP concepts to solve the "classic" pay problem -- read a pay record, compute the pay, and print a report. One additional criteria has been added to the problem: the employee's grade/rank must be between 1 and 30, inclusive.

The structure chart was used as an aid in designing the solution. However, any of the other aids could have been used equally well.

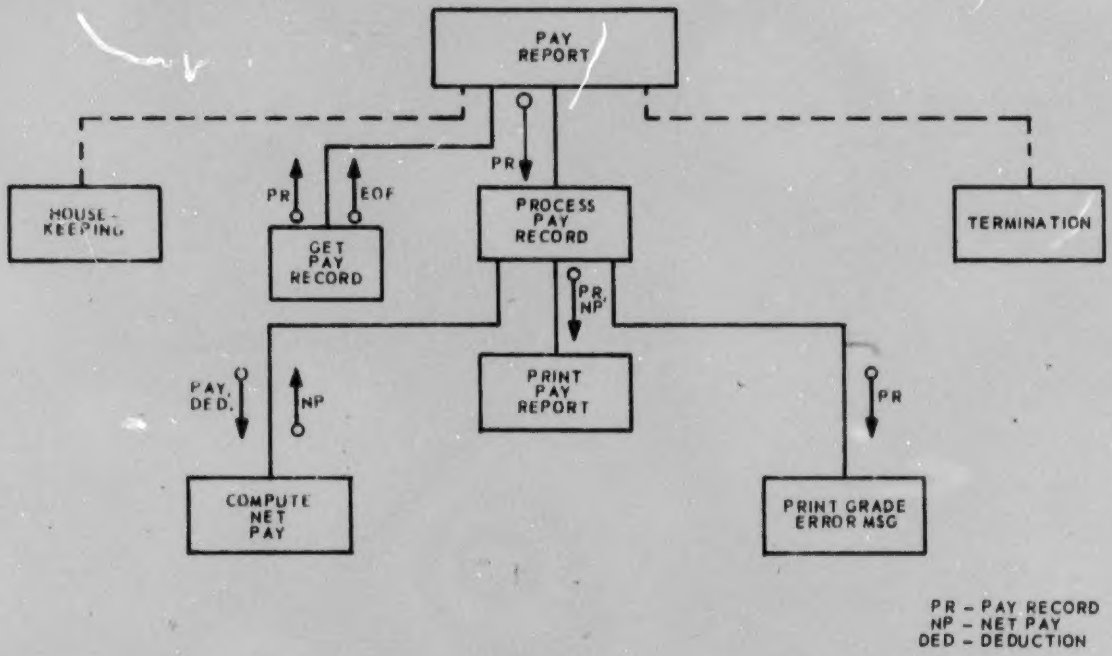


Figure 11-38

The next four figures (11-39 through 11-42) illustrate a possible solution to the pay problem in different languages. Figure 11-39 is a solution in PDL using the sequence, IFTHENELSE, and DOWHILE primitives. In each of the examples, it is assumed that all data items were previously defined.

PDL

```
    .
    .
    .
open files
    .
    .
    .
READ 1st pay record
DOWHILE there are pay records
    IF  1 ≤ grade ≤ 30 THEN
        Compute net pay.
        Print pay report line.
    ELSE
        Print error message.
    ENDIF
    READ  next pay record.
ENDDO
    .
    .
    .
close files
```

Figure 11-39

PAGE NO. | PROGRAM | COBOL DIVISION | PAGE 1 OF 2

PROGRAMMER | DATE | IDENT.

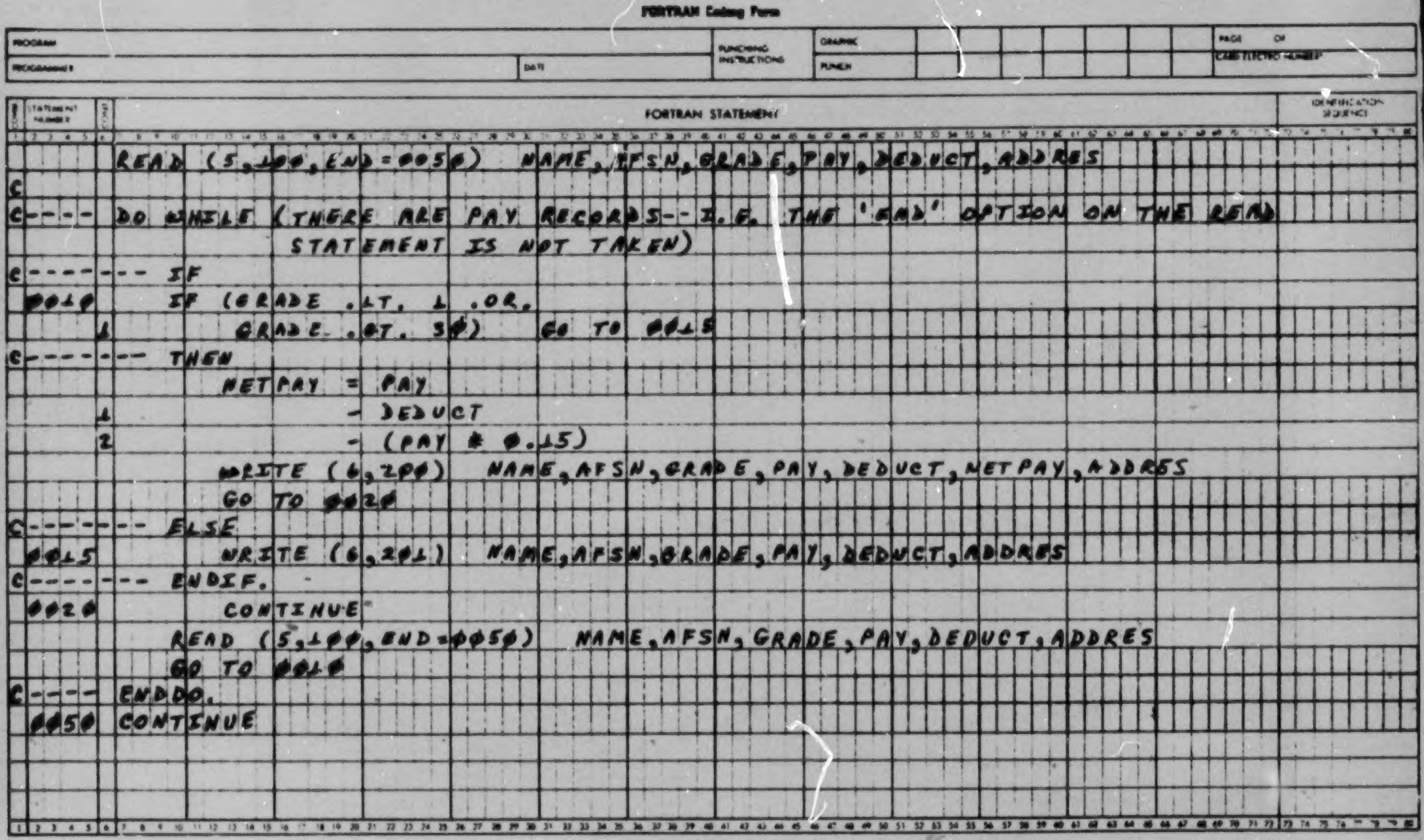
```
           OPEN FILES.
               :
           PERFORM GET-PAY-RECORD.
           PERFORM PROCESS-PAY-RECORD
                   UNTIL E-O-F-FLAG-SET.
               :
           CLOSE FILES.
           STOP RUN.


       GET-PAY-RECORD.
           READ MASTER-PAY-FILE RECORD
               INTO PAY-RECORD
               AT END   MOVE SET TO END-OF-FILE-FLAG.
           EXIT.


       PRINT-GRADE-ERROR-MESSAGE.
           MOVE CORRESPONDING PAY-RECORD TO GRADE-ERROR-MESSAGE.
           WRITE GRADE-ERROR-MESSAGE.
           EXIT.
```

Figure 11-40

PAGE NO.
PROGRAM
COBOL DIVISION
PAGE 2 OF 2
PROGRAMMER
DATE
IDENT.

```
PRINT-PAY-REPORT-LINE.
    MOVE CORRESPONDING PAY-RECORD TO PAY-REPORT-LINE.
    WRITE PAY-REPORT-LINE.
    EXIT.


PROCESS-PAY-RECORD.
    IF GRADE IS GREATER THAN 1 AND LESS THAN 30
     OR GRADE IS EQUAL TO 1 OR 30.
        COMPUTE NET-PAY EQUALS PAY
                                MINUS DEDUCTIONS
                                MINUS PAY TIMES 0.15,
        PERFORM PRINT-PAY-REPORT-LINE;
    ELSE
        PERFORM PRINT-GRADE-ERROR-MESSAGE.

    PERFORM GET-PAY-RECORD.

    EXIT.
```

Figure 11-40 (Con't)


FORTRAN Coding Form
PROGRAM
PROGRAMMER
DATE
PUNCHING INSTRUCTIONS
GRAPHIC
PUNCH
PAGE OF
CARD ELECTRO NUMBER
STATEMENT NUMBER
FORTRAN STATEMENT
IDENTIFICATION SEQUENCE

```
      READ (5,100,END=0050)  NAME,AFSN,GRADE,PAY,DEDUCT,ADDRES
C
C---- DO WHILE (THERE ARE PAY RECORDS--I.E. THE 'END' OPTION ON THE READ
                    STATEMENT IS NOT TAKEN)
C-------- IF
 0010       IF (GRADE .LT. 1 .OR.
     1         GRADE .GT. 30)    GO TO 0015
C-------- THEN
               NETPAY = PAY
     1                - DEDUCT
     2                - (PAY * 0.15)
               WRITE (6,200)  NAME,AFSN,GRADE,PAY,DEDUCT,NETPAY,ADDRES
               GO TO 0020
C-------- ELSE
 0015          WRITE (6,201)  NAME,AFSN,GRADE,PAY,DEDUCT,ADDRES
C-------- ENDIF.
 0020          CONTINUE
            READ (5,100,END=0050)  NAME,AFSN,GRADE,PAY,DEDUCT,ADDRES
            GO TO 0010
C---- ENDDO.
 0050 CONTINUE
```

Figure 11-41

PROGRAMMER

PROGRAM IDENT

PAGE

```
         READ ( = PAYRCD, EOF)         $''READ 1ST PAY RECORD.                ''
''------ DOWHILE   END OF FILE (EOF) IS OFF.                                  ''
           GOTO END001                 $
DO001.
           IFEITH (GRADE GQ  1
               AND GRADE LQ 30)        $''IF GRADE IS BETWEEN 1 & 30, INCLUSIV''
             BEGIN                      ''  THEN------------------------------''
               NETPAY = PAY($0$)        ''NET PAY = PAY (FROM PAY RECORD)     ''
                      - DEDUCT($0$)     ''        - ANY DEDUCTIONS            ''
                      - ( PAY($0$) *    ''                                    ''
                          0.15 )       $''        - (15% OF PAY).             ''
               WRITE (PAYRCD, NETPAY)  $''PRINT PAY RECORD LINE.              ''
             END
''-------- ELSE
           ORIF     1                  $                                      ''
             BEGIN
               WRITER (PAYRCD)         $''PRINT GRADE ERROR MESSAGE.          ''
             END
''-------- ENDIF,                                                             ''
           READ ( = PAYRCD, EOF)       $''READ NEXT PAY RECORD.               ''
END001.
           IF (EOFEQ OFF) $ GOTO DO001 $
''------ ENDDO.
```

Figure 11-42

SUMMARY

Structured programming/coding has often been called "goto-less" programming because an attempt is made to avoid the "GOTO." Structured coding is defined as the coding of programs by repeated use of a selected number of predefined control logic primitives, each with one entry point and one exit. Combinations of the primitives can be used in implementing a program design.

There are three categories of these primitives: the sequential, conditional (IFTHENELSE), and iterative (DOWHILE). Two extensions of these categories are the DOUNTIL (iterative) and the CASENTRY (conditional) figures. These five control logic primitives provide the basis of structured coding. However, there are problems in implementing these primitives.

There are computer languages which do not contain the structured coding control logic figures. Two options for implementing the structured primitives exist. One is to simulate them manually and the other is by using a precompiler. The precompiler accepts the control figures (IFTHENELSE, DOWHILE, etc.) and generates compatible source code for the target language. Even with precompilers, there are problems in structured programming/coding.

The problems in structured coding can fall into two classes: those dealing mainly with the programmer and those with compiler languages. Some of the problems include: difficulty in using structured code to implement a program design, resistance to change, programmer's ego, inefficient machine usage, and using structured code to develop a program design. It takes both the programmer's initiative and abilities to work out solutions to these problems.

In comparing "traditional" coding with structured coding, two things emerge. First, the explicit use of GOTOs has been eliminated for the most part. Second, indentation is used to "block off" functional groups of code. The writing of good structured code will take time.

QUESTIONS

1. Define structured coding.

2. List the categories of primitives.

3. What are the 5 basic control logic primitives? Draw their corresponding flow chart representations.

509

4. What is a precompiler? In what way is it used?

5. (A) Possible problem(s) in using structured coding include(s)

____ a. difficult to implement.

____ b. inefficient machine usage.

____ c. closely related to PDL.

____ d. has only 5 control logic primitives.

____ e. has tendency toward pathological data connections.

6. A characteristic of structured coding includes:

a. There is a limited amount of indiscriminate branching in structured coding.

b. Coding is indented for clarification.

c. If a precompiler is available, structured code can be translated into a form that can be compiled.

d. Any of the above.

505

GLOSSARY OF TDSP TERMS

Abstract Input Data - The data at a point within a problem or its solution which is related to the initial input data and can be viewed as still entering the problem or solution. This data identification is done with a data flow graph for use in the transform analysis strategy.

Abstract Output Data - The data at a point within a problem or its solution which is related to the final output data and can be considered on its way out of the problem or solution. This data identification is done with a data flow graph for use in the transform analysis strategy.

Administrator - See Project Administrator.

Backup Programmer - A peer of the chief programmer who is capable of assuming project responsibility and maintaining continuity of the development effort should the chief programmer become unavailable.

Bottom-up Approach - A technique for coding and implementing a modular design where the lowest level processing modules are completed and tested first.

Bubble Chart - Another name for Data Flow Graphs.

Builds - See Packaging.

CASE - One of the control primitives used in structured programming; PDL term which allows for the execution of one condition from a group of possible conditions.

Central Transform - Those elements of a data flow graph which change abstract input data into abstract output data. The central transform is identified for organizing a modular solution using the transform analysis strategy.

Chapin Charts - See Structured Flowcharts.

Chief Programmer - Technical manager of the Chief Programming Team (CPT), who is responsible for the direction and supervision of team members. Responsible for the complete design of a software system.

Chief Programming Team - Two or more programmers/analysts assigned to a project who possess a combined responsibility for the quality of the delivered project.

Coarse Tuning - The initial step used to avoid a network of crossed lines on an IPO chart. Usually accomplished by resequencing or duplicating items in the input or output.

Cohesion - A measure of the strength of association of elements within a module. Also known as module strength or binding.

Coincidental Cohesion - A module comprised of instructions which have no apparent relationship to each other.

Communicational Cohesion - A module whose elements reference the same data.

Connection - A reference in one module to an identifier defined, declared, described, or otherwise caused to exist in another module. Also known as a data or logical interface.

Construct - A control structure containing one entry and one exit used in a preprocessor programming language to allow the application of structured coding principles in a higher level language. Also known as primitives.

Control Coupling - An association between two modules where flags or switches from one module are used to influence the execution of the other module.

Control Logic Primitives - Building blocks used to construct structured programs. Specifically: Sequence, IFTHENELSE, DOWHILE, DOUNTIL, CASE.

Coupling - A measure of the strength of association between two modules; intermodule relationship.

CPT - Chief Programming Team.

Data Coupling - An association between two modules where information from one module is used to provide input for execution by the other module.

Data Flow Graph - A graphical method of organizing and recording the initial ideas for the solution of a problem which emphasizes the flow of data through the problem.

Decompose - To separate into parts or elements; particularly to divide a task or function into its subfunctions.

Delivery by Parts - See Packaging.

Detail Design Package - The solution to a problem sufficiently specific so that the coding is a simple almost mechanical step in software development.

Detail Diagram - Provides specific information at the lower levels of the hierarchy chart.

Development Support Library - A central repository of all data revelant to the project, in both human-readable and machine-recognizable form.

DFG - Data Flow Graph.

Direct Data Flow - See Normal Data Flow.

DOUNTIL - One of the control primitives used in structured programming; PDL term for iterative execution with the test for loop termination last.

DOWHILE - One of the control primitives used in structured programming; PDL term for iterative execution with the test for loop termination first.

DSL - Development Support Library.

Dummy Module - A temporary unit of source code which is part of an incomplete structured program and will be replaced by the actual unit of code when it is completed.

Factoring - Separating a function into smaller functions. Also called Functional Decomposition.

FAN-IN - The number of higher-level modules calling the same subordinate module.

Fine Tuning - The final graphic step in HIPOs where crossed lines on the IPO charts are eliminated.

Functional Cohesion - A module which performs only one task. See cohesion.

Functional Decomposition - See Factoring.

Global Parameters - Any type of information which is accessible by all parts of a program or system.

Hierarchy Chart - A graphic tool developed during the software design process to illustrate the organization of functions and provide a quick reference for IPO charts. Also called Visual Table of Contents (VTOC).

Hierarchy plus Input-Process-Output - A tool designed to aid in system development by providing a graphical representation from the initial design through the life of the system.

HIPO - Acronym for Hierarchy plus Input-Process-Output.

HIPO Diagrams - See Hierarchy Chart and IPO Chart.

HIPO Package - Consists of overview diagrams, detail diagrams, and Visual Table of Contents (VTOC).

Hybrid Coupling - A combination of data and control coupling where a variable is used for both data and control by the receiving module; also refers to the modification of instructions during program execution (i.e., COBOL ALTER Statement).

Hybrid Parameter - A variable used in Hybrid Coupling.

IFTHEN - A variation of the IFTHENELSE primitive where the ELSE condition is a null function.

IFTHENELSE - One of the control logic primitives used in structured programming; PDL term for a two-state decision mechanism or conditional execution.

Incremental Delivery - See Packaging.

Indirect Data Flow - Information exchange in a modular program or system without being explicit and obvious. Examples are FORTRAN COMMON, COBOL data division, other global variable uses.

Initial Design Package - The first major attempt to develop a software product, therein producing documentation for walk-throughs and for more detailed development.

Interface Complexity - See Coupling.

IPO Chart - A graphic tool used in the software design process to describe and decompose a function in terms of the Input(s), the Process(es), and the Output(s).

Librarian - See Programming Librarian.

Local Parameter - A variable or information item which is created and used only within a specific module.

Logical Cohesion - All elements in the module are involved in similar tasks--such as a general editing routine. See Cohesion.

Maintenance - The effort expended to debug or modify a program which is no longer in the development phase.

Maintenance Package - Any documentation which is deemed useful when modifying or debugging production software.

Module - Functional part of a solution; an assembly, functioning as a component of a larger part.

Nassi-Schneiderman Diagrams - See Structured Flowcharts.

Normal Connection - A module is always initiated at a single entry point and terminated at a single exit point. Control always returns to the point where a module was invoked. Refers to modules by name with no direct reference to that module internally.

Normal Data Flow - Passing data from one module to another by explicitly stating the variables in conjunction with a normal connection.

Normal Transfer - See Normal Connection.

N-S Diagram - See Nassi-Schneiderman Diagram.

Overview Diagram - Presents a general description of a function; usually supplements the hierarchy chart (VTOC). Lists general inputs, processes, and outputs without showing relationships.

Packaging - A tool used to determine where to divide the problem for development; the art of subdividing a skeletal design into several parts or packages. (Also called Builds, Incremental Delivery, or Delivery by Parts.)

Pathological Connection - A module reference to an internal program element (either data or control) in another module; an abnormal exit from, or entrance to, a module.

Pathological Data Flow - See Indirect Data Flow.

PDL - Program Design Language.

Physical Data - The information available prior to and after system or program execution. Generally thought of but not limited to punched cards, printer output, magnetic tape, disk, etc.

Precompiler - A program which is invoked before the compiler to convert those extensions of the source language, which support structured programming, into acceptable source language code.

Primitives - See Control Logic Primitives.

Production Library - Central repository for all codes that have been tested, and integrated with all higher level codes; constitutes the current operational system.

Program - Logically the same as a system or module, but generally, a system consists of programs.

Program Design Language - An English-like language for describing the control structure and general organization of a computer program.

Program Stubs - Segments of code which substitute for modules not yet developed so that a higher level completed module can be tested using the stubs in place of the incomplete lower level called modules.

Programming Librarian - An integral member of the Chief Programming Team (CPT) who develops and maintains the Development Support Library (DSL) and Programming Support Library (PSL).

Programming Support Library - A software system which provides the tools to organize, implement, and control computer program development.

Project Administrator - A support function for the nontechnical requirements of a Chief Programming Team such as scheduling, budgeting, and personnel requirements.

Project Manager - The individual ultimately responsible for the development effort. The supervisor of the chief programmer, often performing the function of project administrator.

PSL - Programming Support Library.

SAPTAD - Acronym for System Analysis, Program, Transaction, Action, Detail. Associated with transaction analysis.

Sequence - A basic control logic primitive which is recognized as the execution of one statement after the other, in order.

Sequential Cohesion - Module in which the input for one module element is output from a previous module element. See Cohesion.

Structure Chart - A simple, flexible design tool which is used to organize and document the thought process leading to a problem solution.

Structured Coding - The writing of programs by repeated use of predefined control logic primitives: Sequence, IFTHENELSE, DOWHILE, DOUNTIL, and CASE.

Structured Design - An approach to software development with tools and techniques intended to reduce complexity and facilitate maintenance.

Structured Flowcharts - A detailed design tool used to specify the steps required to perform a function in terms of specific primitives.

Structured Programming - The process of developing structured programs; associated with structured programming are certain practices such as indentations of source codes to represent logic levels, the use of intelligent data names, and descriptive commentary.

Structured Walk-Throughs - Technical examinations of the design, implementation, and documentation to provide positive feedback to the programmer. (Or a set of formal procedures for reviews--by the entire programming team--of programming specifics, programming design, actual code, and adequacy of testing.)

Superordinate - Opposite of subordinate; boss or superior; with modules, the calling module is a superordinate of the called module.

Support Members - Individuals required by the Chief Programming Team to provide knowledge and expertise in a specific area; i.e., other programmer, contract officer, accountant, project administrator, consultant, etc.

TDSP - Acronym for Top Down Design and Structured Programming.

Temporal Cohesion - Module containing time related elements, such as initialization and termination routines. See Cohesion.

Top Down Design - The technique of developing the framework for the solution to a problem by repetitively factoring the problem into an organization of small problems which can be more effectively solved.

Top Down Documentation - Illustrates the top down design and is delivered in increments as the system is developed. Includes descriptions, specifications, graphical representations, manuals, plans, reports, listings, etc.

Top Down Implementation - The coding, verification, and implementation of higher levels of the system logic prior to the coding of any subordinate modules; an approach to the design, coding, and testing of the system.

Top Down Structured Program - A structured program with the additional characteristics of the source code being logically segmented in a hierarchical manner and only dependent on codes already written.

Transaction Analysis - A strategy for designing highly modular programs and system by organizing all the combinations of events which are possible in processing a particular data item.

Transform Analysis - A strategy for designing highly modular programs and systems by studying the data flow through the problem.

Tuning - The process of avoiding crossed lines in the graphics of an IPO chart. See Coarse Tuning and Fine Tuning.

VTOC - Visual Table of Contents - See Hierarchy Chart.

TRAINEE'S COMMENT FORM

TDSP Student Text

Please help us improve our training literature by sending us your opinions, suggestions, etc. We are especially interested in receiving your comments on the readability of this text, recommended additions and deletions, and identification of any errors.

COMMENTS

---- Fold ---- Fold

---- Fold ---- Fold

FOLD ON TWO LINES, STAPLE AND MAIL

517

---- Fold

---- Fold

3390 Technical Training Group/TTMKP
Keesler Air Force Base, MS 39534

Attn: Curricula Development Unit

---- Fold

---- Fold

MISNUMBERED- P. NO. 518 MISSING

STUDY GUIDE
E30BR5141 002
KDA-478

Technical Training

Computer Systems Programming Officer

PROGRAMMING PRINCIPLES

February 1977

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC Keesler 9-8339

DO NOT USE ON THE JOB

SG E30BR5141-002
KDA-478
February 1977

PROGRAMMING PRINCIPLES

CONTENTS

TITLE	PAGE
INTRODUCTION	1
PROBLEM SOLUTION METHODS	2
Direct Method	2
Enumerating Method	3
Scientific Trial and Error Method	3
The Simulation Method	3
THE ALGORITHMIC STATEMENT OF PROBLEM SOLUTIONS	4
The Direct Algorithm	5
The Repetitive Algorithm	5
The Indirect Algorithm	5
PROBLEM DEFINITION	5
The Input-Process-Output (IPO) Chart	7
The Hierarchy Chart	11
FLOWCHARTING THE PROBLEM SOLUTION	15
The Sequence Primitive	16
The IF...THEN...ELSE... Primitive	17
The Case or Switch Primitive	20
The DO UNTIL Primitive	25
The DO WHILE Primitive	27
The LOOP EXIT IF Primitive	29
Miscellaneous Flowchart Symbols	30
General Guidelines for Flowchart Preparation	30
FLOWCHART DEVELOPMENT	31
Arrays and Subscripting	36
Sequential Search Algorithm	36
The Bubble Sort Algorithm	44
DEVELOPMENT SUPPORT LIBRARY	48
TEAM OPERATIONS	50
STRUCTURED WALK-THROUGHS	53
Mechanics	54
As Part of New Technologies	55
Parallel Testing	57
Psychology	58
THE IDEA OF STRUCTURED PROGRAMMING	59
Top Down Programming	59
Structured Programming Theory	59
Segmenting Structured Programs	61
Creating a Structured Program	62

SC E30BR5141-002
KDA-478
February 1977

PROGRAMMING PRINCIPLES

INTRODUCTION

Computer programming may be defined as the specification of a set of instructions, in a form acceptable to a computer, prepared in order to achieve a certain result. This statement sounds just like a high school definition that will surely show up on a test at some time, but it will not be on any test in this course. We are going to break that definition into three parts, define a meaning for each part, and look at them separately.

The first part we will describe is . . . "The specification of a set of instructions . . ." We will call this the Problem Definition or solving the problem. Quite often when a problem is defined in detail, the problem solution becomes evident. The problem's definition may be specified in any language, e.g., algebra, set notation, English, COBOL, FORTRAN, or Swahili. A problem definition may be thought of as an outline of a book--all the important points are listed in the order they are encountered. In programming circles this may also be referred to as a "program design."

The second part of the description we will discuss is ". . . in a form acceptable to the computer . . ." This is the actual written computer program. It must be written in a machine acceptable language, e.g., machine, BAL, COBOL, ALGOL, FORTRAN, or RALF, etc. This is the step in which you translate the problem definition into a language the computer can understand.

The last part of the definition, and by many standards the most important, is ". . . in order to achieve a certain result." This is the problem solution. It is the result of the design of a computer program. Without getting a solution to the original problem, we can in effect say that nothing has been done in the hours, days, or months it has taken to design and code a program. If our program does not achieve its desired result, there can only be two causes. Either the program design did not correctly solve the problem, or the problem design was not correctly translated (coded) into the machine acceptable language.

The first step in writing a computer program is considered by many to be the most difficult part of programming. In fact, this is the step that separates the "coders" from the "programmers." A programmer is a person capable of properly defining a problem, designing a program to solve the problem, and coding that program. A coder can only code a program in a machine acceptable language from someone else's design. When you finish this course, each of you will have the knowledge necessary to become computer programmers. What you do with that knowledge will determine your capabilities and growth as a computer programmer.

In this study guide we will concentrate on problem solving techniques and program design. Remember, a correct program design is really an ordered list of all processes you would have to perform if you were the computer. If a design does not contain enough information for you to solve the problem, it is certainly not detailed enough for you to tell an "idiot with the speed of light" (the computer) how to solve the problem.

PROBLEM SOLUTION METHODS

Selecting and analyzing problem solution methods are used throughout the problem definition phase of computer programming. It would be helpful at this point to discuss these two areas of problem solving before we begin examining ways to define problems in detail.

Solving a problem is the process a person goes through to define a process which will give a desired unknown result by using known information or equipment. Unfortunately, there is no basic technical school wherein one can be taught how to solve problems. Some people are better at it than others. This leads to the idea that problem solving is an intuitive, or unconscious, process.

If this is true, then a person cannot be taught how to "solve a problem," but he can be shown some of the methods used by the good problem solver. He then may be able to consciously apply these methods to his defined problem in order to arrive at a solution.

It can be shown that even though a person may solve different problems in what appears to him to be the same way, he in fact gets to the solution in one of, or a combination of, the following different methods.

- DIRECT.
- ENUMERATING.
- SCIENTIFIC TRIAL AND ERROR.
- SIMULATION.

In all the methods we will examine, the first thing to be done is to put the problem into a proper framework. This framework consists of three parts.

1. Identify the known parts and their relationship to the solution.
2. Identify the solution.
3. Identify the unknown or variable parts.

Having done this, it becomes a matter of building a string of relationships between the known parts of the problem and the solution to the problem.

Unfortunately, it is difficult to know which relationships will lead most directly to the solution. If a person moves efficiently from known parts to solution, he probably has more "feel" for the problem than one who stumbles about before finding the right path.

Direct Method

The direct method is probably the most widely used of all problem solving methods and, therefore, it is so ingrained and habitual that it appears to be purely intuitive. For this reason, it is extremely difficult to explain or to analyze it scientifically. If you consider being thirsty a problem, then the solution to the problem would be "go to the water fountain and get a drink." This would be an example of a direct solution method.

Enumerating Method

This method is also widely used and is so simple that it might seem that there is little reason for even listing it. It consists of checking every possible entity that could be a solution.

In other words, if one wished to find the heaviest book in a library, he would weigh each one, check each weight, and select the book with the largest number. There are two requirements for using this method.

1. All of the solution possibilities must be known.
2. There must be some criterion against which to match the possibilities.

Scientific Trial and Error Method

Similar to the enumerating method, the scientific trial and error method is used where the number of possible solutions is very great. In practice, it works this way.

A guess is made at the answer. We try to come as close to the solution as possible, but it is not necessary to be close. Then the guess is examined for what it has done for the problem. Based on the results of this examination, another guess is made. Hopefully this is a better guess, and we come even closer to the solution. These steps are repeated until the solution is found. This may at first appear to be a very primitive approach, but it is a very versatile and useful method of attacking a problem; it is well suited to computers as it is basically a reiterative method.

An excellent use of scientific trial and error would be a common method for taking the square root of a number using a basic 4 function calculator. Let's call the number we want to take the square root of "X." Then we will let our initial guess called "G" be X divided by 2 ($G = X/2$). Now, using the language of algebra:

$$Y = (X/G + G)/2$$

Now if Y and G are equal, or very close together, we are through; otherwise, we let G equal Y and then reevaluate the equation.

The Simulation Method

The simulation method of problem solving goes a step beyond the mathematical model of our algorithm. With this method we attempt an actual working model of the real world. The working model is set in motion and we sit back to see what actually happens.

As an example, let us help the town of Loannisport with a traffic problem. The main intersection of the town has a drastic problem every day at 5:30 P. M. Traffic seems to be tied up in all directions for more than an hour.

The town council has decided that a more modern traffic signal will get things moving quicker, and the problem to be resolved is the timing and display sequence of the light.

Using the simulation method here seems to be in order. Data on traffic flow is gathered, along with other data that may influence the problem—such as the number of pedestrians using the intersection, schools in the area, maximum speed limits consistent with safety, etc. A mathematical algorithm is built with this data and a variable.

The variable is the various timing and display sequence which causes the least congestion at the intersection. The algorithm is written for the computer, and the program is run using different values for the computer to give us the answers.

The last problem gives us the opportunity to make a very important point. A solution and answer are two different entities. Note that in Loannisport the solution to the traffic problem was the installation of a properly sequenced traffic light at the main intersection.

An answer relates to a specific set of problem specifications where a variable has been given a value, and that value is used in computation. Every time a different sequence for the light was used in the mathematical algorithm, the computer came up with an answer. Only one, or at most only a few answers, satisfied the specifications of the solution, which was a minimum amount of congestion at the intersection.

One other thing we should point out here is that it is very difficult to pin a label on the method of attack for a particular problem. Any one problem may involve all of these methods to some degree, even if the solution comes so quickly as to defy naming the process. If we go back to the problem of quenching a thirst, we said the solution used the direct method; however, the process of going to the water cooler was a repetitive process of putting one foot in front of the other. This could be considered to be scientific trial and error, because each time you take a step you reevaluate your direction of travel based on your present position with respect to the water cooler.

THE ALGORITHMIC STATEMENT OF PROBLEM SOLUTIONS

An algorithm is the set specifications or instructions for doing something. Usually that something is solving a problem. As such, an algorithm need not be mathematical. A mathematical algorithm might be as follows: Take a number, multiply it by itself, add to it twice itself, then subtract one and call this the result. This English algorithm can be stated in another language called algebra.

$$R = X^2 + 2X - 1$$

A good algorithm has two properties. First, it must be clear. Each step must have one, and only one, interpretation. Everyone reading the step must be able to accomplish it in the same way, arriving at the same result. Second, an algorithm must stop.

To demonstrate the importance of these two properties, take as an example an algorithm for making a good cup of coffee, in which the fifth step is written: "Add 1 teaspoon of sugar until the coffee is sweet enough." We have said that the algorithm must be clear and must stop, but the concept of sweetness varies from person to person; thus, a person reading these instructions who has a deformed sense of taste may never stop adding sugar to the cup. A better fifth step would be "Add one teaspoon of sugar" followed by step six, "Taste coffee"; step seven, "If it is not sweet enough, add another teaspoon of sugar"; and step eight, "Repeat steps six and seven until the coffee is sweet enough or ten teaspoons of sugar have been added." Note that this modified algorithm is clearly stated and will stop, but it may not solve the problem of making a good cup of coffee for the person with a deformed sense of taste.

For the majority of our work we, of course, are interested in algorithms that solve our problem. However, not solving the problem does not invalidate the algorithm; it must only be elementary, clear, and terminating.

Another algorithm might be:

1. Remove the water from a pond with an 8-ounce glass.
2. Stop when the pond is empty.

This is a straight-forward, unambiguous task. We can be sure also that it will eventually stop, even though we do not know how long it might take.

The Direct Algorithm

A direct algorithm is one that is made up of a number of known steps and a result determined by these steps. The first example we gave you in the previous section is a direct algorithm. We can tell just by looking at the wording or the algebraic notation, $R = X^2 + 2X - 1$, just what and how much is involved in obtaining a result.

The Repetitive Algorithm

A repetitive, or iterative, algorithm is one in which some or all of the steps of the process are repeated. An example of an iterative algorithm was given in the example of making a good cup of coffee in which step 6, "Taste the coffee," and step 7, "If not sweet enough, add another teaspoon of sugar," were repeated until the coffee was sweet enough or ten teaspoons of sugar had been added.

The Indirect Algorithm

Very simply, an indirect algorithm is one that is not direct. We have said that a direct algorithm has a number of known steps, and we could tell how much work had to be done by looking at it.

Note the second example where we referred to removing water from a pond with an 8-ounce glass. It, like the coffee algorithm, is iterative but it is unknown at the beginning how many times the process of dumping one glass of water must be repeated. It will be seen that the indirect algorithm can be put to great advantage in solving problems with the computer.

PROBLEM DEFINITION

In the introduction to this study guide, we stated that "Quite often when a problem is defined in detail, the problem solution becomes evident." You have probably already asked yourself some questions about this statement, such as: What do we mean by define in detail? How can we define a complex problem in detail? Why not just do the job and define the problem as we go? These are all valid questions that will be answered as we progress through this section.

First, why even worry about defining the problem? To illustrate the need for a complete problem definition, let us suppose you were given some Air Force orders sending you on temporary duty. The orders stated only:

Report to: Colonel Arnold Flakbait
123 S. Main St., Rm 461
Washington

at 0800 hours 3 days from today.

Do you have the information you will need to perform the job you have been given? No? You are right. How are you going to get there? Plane? Car? Train? Stagecoach? What do we mean by today? Now? The day the orders were printed? The day Colonel Flakbait said "Send G.I. Joe to Washington?" For that matter, Washington where? DC? State? Arkansas? Suppose you get the dates straightened out, discover you are to drive, and Washington is in the Northeast United States. What do you do? Sit down and decide which highways you have to take to get to Washington, DC? (You read the Air Force Times and know Col Flakbait can usually be found hanging around the Pentagon.) I hope you don't go to DC because Col Flakbait is expecting to meet you in Washington, Conn.

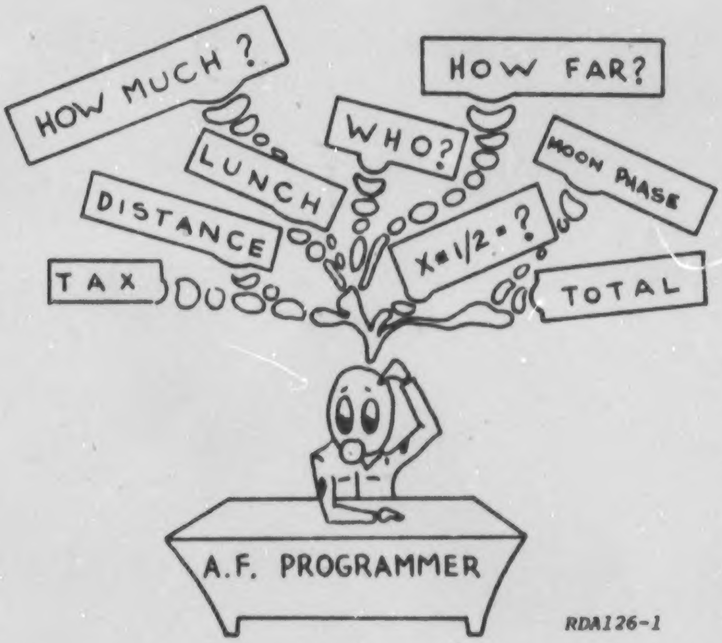
What? You say the Air Force would not issue any orders like the one above? True! However, you may rest assured that as long as you are a computer programmer, people will bring you problems that are not even as well defined as our proposed TDY orders.

Consider, for example, the businessman who wanted to write a program to computerize his stock inventory. His program consisted of:

> "Dear Computer,
>
> Please take the following information and save it so that you can give it back to me in the way I want it when I want it."

Our businessman followed this statement with his store's inventory list. He was quite serious and felt he had given the computer all the information it needed to do what he wanted done. Do you know what he wanted? I don't!

We will resort once more to a definition here--not to be memorized but to be sure that each of us is talking about the same thing. Webster defines PROBLEM this way: A question proposed for solution or consideration. If we do not take too much notice of the last two words for the present, we have a working definition of our purpose as computer programmers--to propose solutions to a question. The question, therefore, becomes the problem.



RDA126-1

However, before we jump into writing a program to solve a given problem, there are a number of things that must be done. One of the things to be done at the beginning is to eliminate as many assumptions as possible. Assumptions will probably get you into more trouble than anything else. Do not assume that you know what the problem is on the first telling or reading. Unfortunately, the language used to relate the problem to the programmer is usually English (which can have infinite shades of meaning to different people). This is bad enough, but add a smattering of technical jargon and who can be sure what is meant. Remember the note from the boss to his employees? "I'm sure that you think you understand what you thought I said, but what I meant to say is not what I think you have assumed I meant."

Now that we are well aware of the pitfalls that can be found in the realm of problem solving, let's see if we can find a method that will help us to steer around these pitfalls.

The first rule is to write everything down! Don't rely on your memory because little things will slip by you, or be forgotten. If you don't believe me, what room number is Colonel Flakbait going to be in? No fair peeking! The reason you had trouble remembering the room number is because it didn't seem important when you read it. Very often, facts that seem insignificant or unimportant when the problem is first presented to you can turn out to be the vital key to the successful solution of the problem.

The second rule is to solve only simple problems. However, you and I both know that people are not going to bring you simple problems to solve--they will solve those themselves. You are going to be given some problems that will put you into the "mumble mode." If we are given complex problems and all we can solve are simple problems, what do we do? Obviously we must somehow make the complex problem simple. Easier said than done, you say? Agreed. But if we were to examine a complex problem closely, we would discover that it is made of many simple problems. It is not likely, however, that we will be able to use the direct method and just write down all the simple problems that make up our complex problem. A Scientific Trial and Error method might be appropriate. We begin by breaking our complex problem into a few simpler problems. If you divide a complex problem into smaller subproblems, each subproblem must be simpler than the whole problem. Each subproblem only contains a part of the problem--much like breaking a pencil in half, each piece is smaller than the whole but, together, the makeup is whole.

As you are reading the next few pages, you may get the feeling "This is ridiculous. Anyone in his right mind would just start getting the answer to the problem and making any decisions as he comes to them." True! That will work fine for you because you have the unique ability to reason. However, the computer does not have this ability. You, as a programmer, are required to find and solve all possible alternative solutions to any problem for which you wish the computer to calculate an answer.

The Input-Process-Output (IPO) Chart

Consider now a workable method for subdividing a complex problem and for writing everything down. Take a piece of paper, briefly state what has to be done at the top, and then divide the remaining part of the page into three columns. In the right-hand column, state the specific result or results of the solved problem. This is known as the output of the problem solution. In the left-hand column, write down all the information you have pertaining to the problem. This is called the input to the problem solution. In the middle column list all of the things that have to be done to get from the basic known information in the left-hand column to the desired result(s) in the right-hand column. We will call this column the process column.

As an example, let's go back to our original set of vague orders to go TDY. That is a problem! What has to be done? Go TDY? Right. But what specific result do we need? Of course--"Report to Colonel Flakbait." In the process column, list the major tasks that must be performed to produce the result(s) listed in the output column.

GO TDY

INPUT	PROCESS	OUTPUT
Col Arnold Flakbait 123 S. Main St, Rm 461	Gather needed information	Report to Col Flakbait
Washington?	Plan trip	
Need to go TDY	Travel	
Base Personnel Office		
Base Transportation Office		

RDA126-2

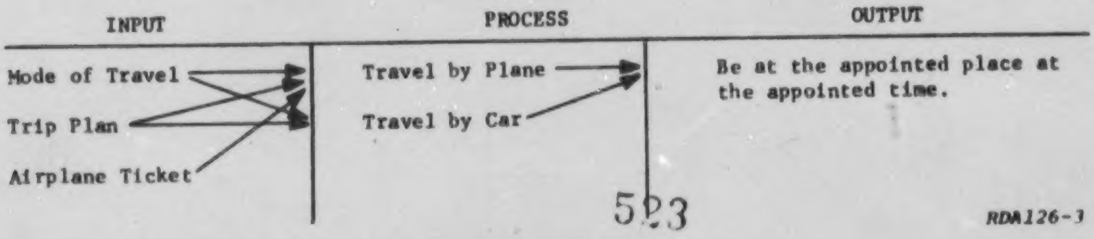
You will note that some items other than the information on your orders are listed in the input column. This is because it is always better to have too much information than not enough. If something is listed in the input section that was not used to solve the problem, it can merely be crossed off. Not so with information omitted. It is necessary at this point to emphasize that the tasks listed in the process column, when performed, must produce the results listed in the output column. Don't worry now about how you are going to perform each process--that will come later.

We now have a complex problem that has been subdivided into three simpler problems. If you can easily see how to solve any of the tasks listed in the process column, then they are simple problems. Solve them! Treat each of the remaining tasks as a complex problem, subdividing them using the IPO chart as discussed above.

In the text where we discussed solution methods, we stated that sometimes it is helpful to work backward from desired result to known information. This is an appropriate method of attack for this problem. We will begin with the travel process.

What is the desired result of our travel? To be at the appointed place at the appointed time? Right. Okay, what information do we need to perform the travel function or process? We need to know the mode of travel and have a trip plan. (We assume the trip plan will be valid to get us to the right place at the right time.) Our process for the travel function then would be to either travel by car, or travel by plane. (The Government doesn't use stagecoaches much anymore.) But if we travel by plane, we will need a ticket, so let's add that to our input column. Our IPO chart for the travel function now looks like this:

TRAVEL

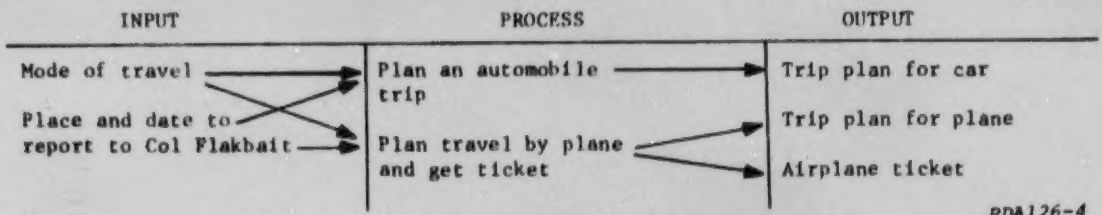


INPUT	PROCESS	OUTPUT
Mode of Travel	Travel by Plane	Be at the appointed place at the appointed time.
Trip Plan	Travel by Car	
Airplane Ticket		

RDA126-3

Notice the addition of the arrows. In this situation, it is relatively easy to see which inputs are used by which process and which process results in which output. We draw the arrows so that the next time we look at the IPO chart we won't have to waste time analyzing and making a decision that has already been made. Write everything down, remember? We will continue the process with the trip plan task.

TRIP PLAN



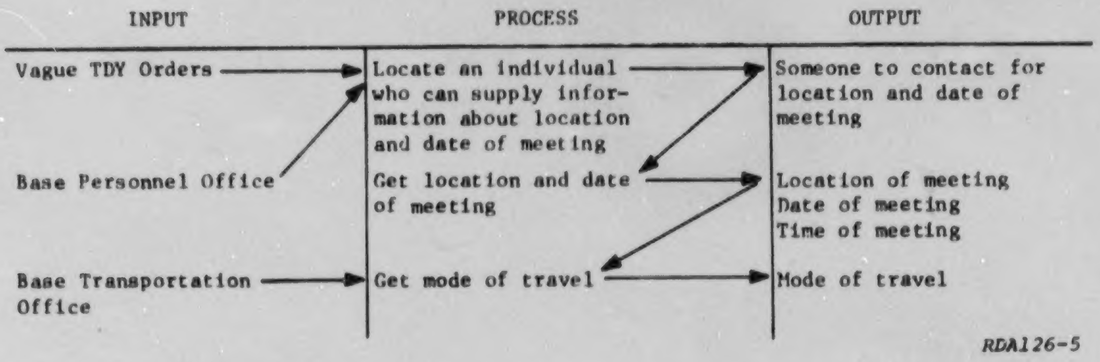
INPUT	PROCESS	OUTPUT
Mode of travel	Plan an automobile trip	Trip plan for car
Place and date to report to Col Flakbait	Plan travel by plane and get ticket	Trip plan for plane Airplane ticket

RDA126-4

Notice here that two different results are listed in the output column, and that each output was the result of performing a separate process. This fact will be used later.

Now we will repeat the process for the "gather needed information" task.

GATHER NEEDED INFORMATION



INPUT	PROCESS	OUTPUT
Vague TDY Orders	Locate an individual who can supply information about location and date of meeting	Someone to contact for location and date of meeting
Base Personnel Office	Get location and date of meeting	Location of meeting Date of meeting Time of meeting
Base Transportation Office	Get mode of travel	Mode of travel

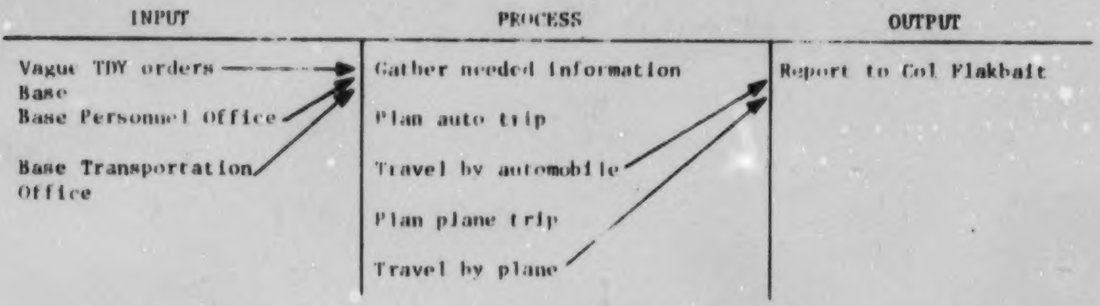
RDA126-5

Something new has been added. An output from one process has been used as an input to another process within the same general problem area or module. This is perfectly all right. The arrows indicating this will be a big help later when we are deciding the order in which the processes (functions) are to be performed.

We have now completed subdividing our complex problem one level. We have also defined each of the subproblems as a module which contains one or more functions to be performed. This may or may not be sufficient to determine a solution method for each subproblem. In most cases it will not. However, before we go on with the subdividing process, let's take a close look at what we already have. Several places we used different words to express the same meaning (shame on us). This is not a good idea because it could cause confusion later on. We should go back and rewrite our IPO charts, using the same words to express the same meaning. Before we do that, however, are there any IPO charts that list two or more seemingly independent processes? Sure, the "trip plan" and the "travel" IPOs seem to have independent processes. Let's separate those and make each process a separate module or subproblem of our main problem.

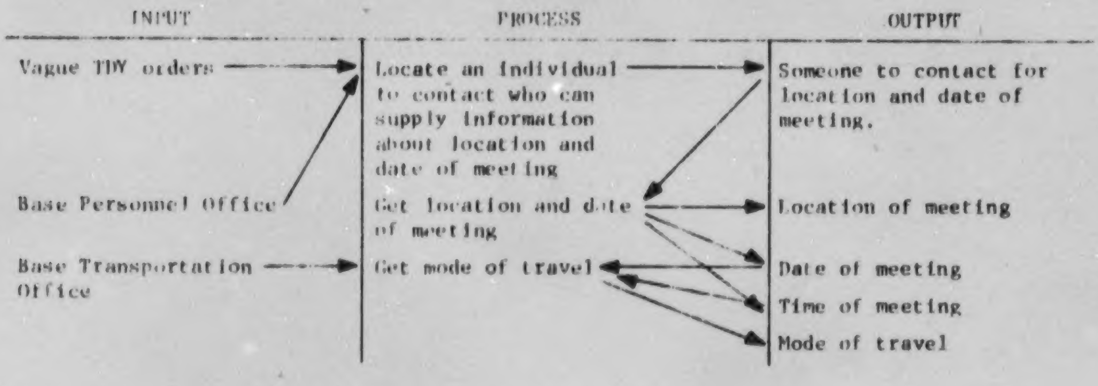
Our IPO charts now look like this:

GO TDY



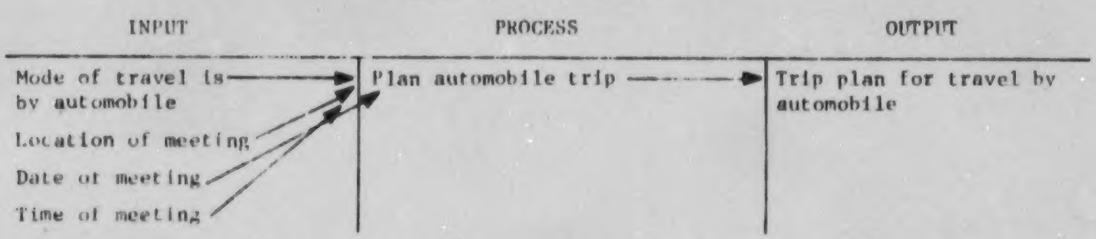
INPUT	PROCESS	OUTPUT
Vague TDY orders Base Base Personnel Office Base Transportation Office	Gather needed information Plan auto trip Travel by automobile Plan plane trip Travel by plane	Report to Col Flakbait

GATHER NEEDED INFORMATION



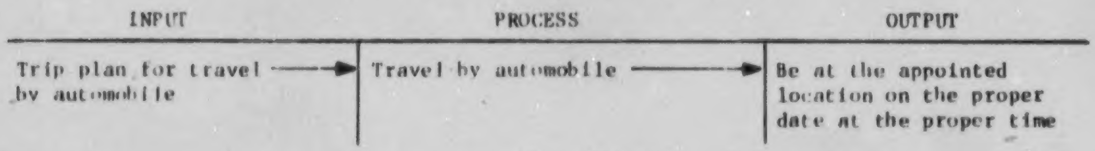
INPUT	PROCESS	OUTPUT
Vague TDY orders	Locate an individual to contact who can supply information about location and date of meeting	Someone to contact for location and date of meeting.
Base Personnel Office	Get location and date of meeting	Location of meeting
Base Transportation Office	Get mode of travel	Date of meeting Time of meeting Mode of travel

PLAN AUTOMOBILE TRIP



INPUT	PROCESS	OUTPUT
Mode of travel is by automobile Location of meeting Date of meeting Time of meeting	Plan automobile trip	Trip plan for travel by automobile

TRAVEL BY AUTOMOBILE



INPUT	PROCESS	OUTPUT
Trip plan for travel by automobile	Travel by automobile	Be at the appointed location on the proper date at the proper time

PLAN PLANE TRIP

INPUT	PROCESS	OUTPUT
Mode of travel is by plane → Location of meeting → Date of meeting → Time of meeting →	Plan a plane trip →	Trip plan for travel by plane

TRAVEL BY PLANE

INPUT	PROCESS	OUTPUT
Trip plan for travel by plane →	Travel by plane →	Be at the appointed location on the proper date at the proper time

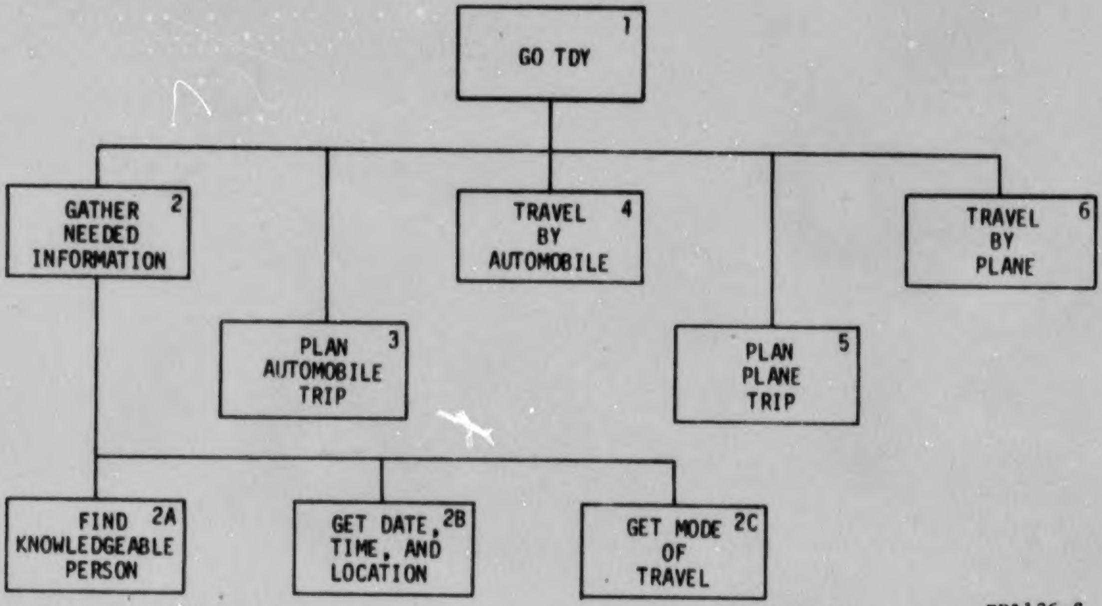
RDA126-7

You may have noticed that the previous four IPO charts listed the same process that was listed as a subproblem of our overall complex problem. This should only be done if a solution method is readily apparent to anyone who might be studying the IPO charts. This is obviously not the case. You guessed it! Subdividing these four problems will be left as an exercise for you to do. You may find that some required items have not been listed. If so, add them to the input column of the appropriate IPO chart. For example, you would probably need an automobile before you could drive to your destination. For that matter, would you want to plan an automobile trip if you didn't have a car available? Keep in mind, however, that whenever an item is added to the input column of one module, then some corresponding entry or entries must be made elsewhere in your problem description to insure the new required input item will be available when needed.

Did you notice that every time we subdivided a problem, more questions seemed to appear? This is normal. The questions were always there, we just were not aware of them. A problem is not completely defined as long as some question remains unanswered, or some assumption has not been accounted for. By this definition, none of our subproblems have been completely defined. They have, however, served their purpose of showing a need for, and a method of, dividing a problem into less complex subproblems to enable the development of a solution.

The Hierarchy Chart

You can readily see that as we continue this subdividing process we are going to produce a lot of papers, each with an IPO chart on it. We already have six and are nowhere near complete with our description. It might be helpful to devise a method of indexing our IPO charts so that we can keep some order to our "documentation" of the problem solution. Have you ever heard the statement, "A picture is worth a thousand words"? We believe that statement, so we are going to draw a picture of our problem solution. We will call this picture a hierarchy chart. It will be used to show which simple problems are parts, or components, of more complex problems. It is very similar to an organizational chart which shows which office is responsible to which office. We will start by listing the main problem statement in a box at the top center of the page. Each process listed in the IPO chart for the main problem statement will be listed in a separate box in a horizontal line below the top box. If the second level IPO chart for a particular subproblem contains more than one task or process, then each process is listed in a box below and connected to the box that contains a description of the subproblem we are working with. If that seems a bit confusing, it is because we used words to describe a picture. The example below should help clarify things.

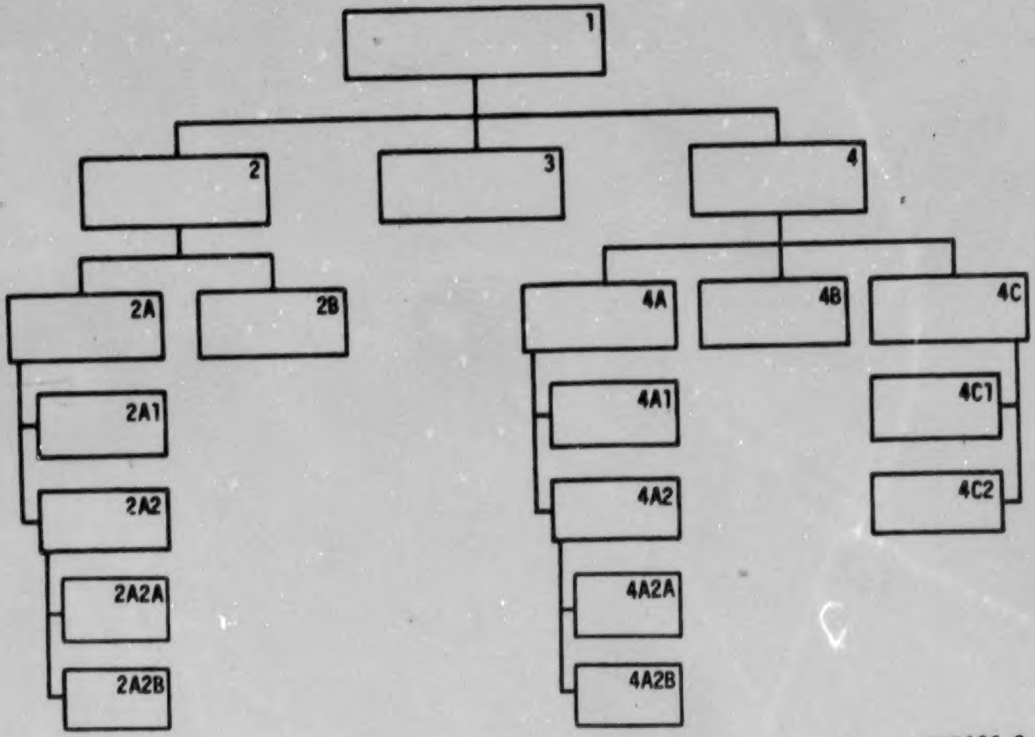


RDA126-8

What? You wonder what to do if you have too many processes to fit on a page? Well, take the subproblem "Plan Automobile Trip," for example. Obviously, if that subproblem had several processes listed, the chart could get very messy and hard to read. We simply draw another hierarchy chart with "Plan Automobile Trip" listed in the top box and its processes listed on the second level of that hierarchy chart. This type of repetition may be continued as often as necessary. Our hierarchy chart gives us a bird's eye view of our complex problem and its component subproblems. But, how will that help us keep out IPO charts in some kind of order?

Did you notice the numbers and letters in the upper right-hand corner of each box? These numbers and letters are a "key" to a verbal index that will be developed in a moment. First, we want to see how these numbers are assigned.

The top center box is numbered 1 on the hierarchy chart that shows the breakdown of the overall problem. The second level is numbered from left to right starting with 2. On the third level, each box is given the number of its associated second level subproblem, and then a letter is attached to the number, giving a unique designation to that particular module. The same letter may be used to designate different modules on the same level if, and only if, they are processes of different higher level modules. For example, if the module "Travel by Plane" had contained three separate processes, the boxes in the hierarchy chart showing these processes could have been numbered 6A, 6B, and 6C. On the fourth level, another number would be added; on the fifth level, another letter, and so on. See the chart which follows.

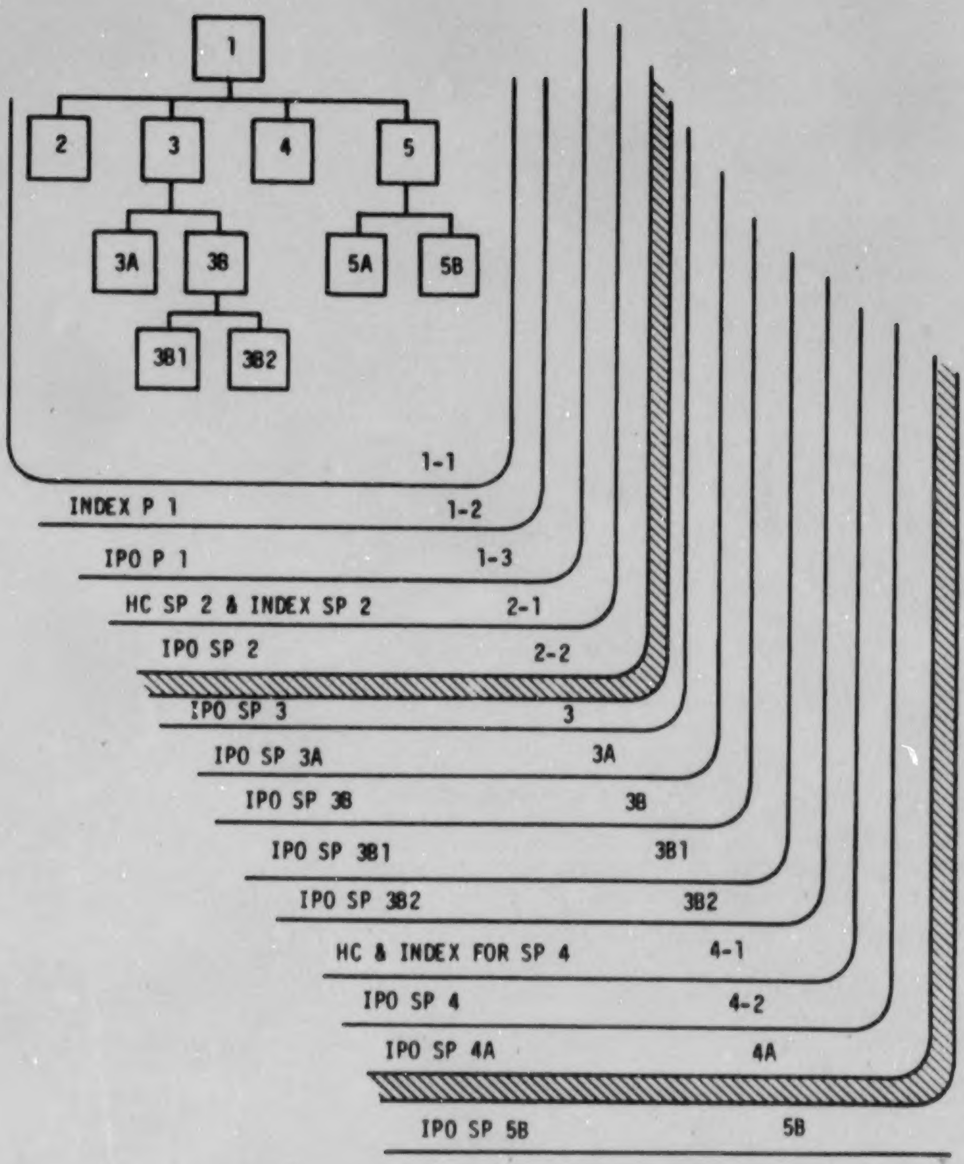


RDA126-9

If more hierarchy charts are needed to show the division of the overall problem, then the top box of these subsequent hierarchy charts will be given the same number that was assigned to this subproblem or module on the earlier hierarchy chart. The numbering system then continues on (and on and on). In this way, we insure that no two subproblem statements will ever be assigned the same key number.

Now that we know how to draw a hierarchy chart, let's look at a plausible means of linking that hierarchy chart to our many IPO charts. A verbal index should do the trick. What two things should an index contain? Right! A short description of an item and the location of the item. The index for our hierarchy input-process-output (HIPO) package may be either on the same page as our hierarchy chart, or it may be on the page immediately behind the hierarchy chart. The index would be easier to use if it were on the same page as the hierarchy chart because you could see everything at one glance.

We will use the "key" number and the verbal description from the hierarchy chart as the description for our index. Besides this, we will list the page number in the HIPO package where the appropriate IPO chart or the more detailed hierarchy chart can be found. Since our key values are unique for any module, it would seem logical to use these "key" numbers as page numbers for our HIPO package. This will work out fine except for one little "fly in the ointment." The page number of our first hierarchy chart could be 1-1, the page number for the index for the hierarchy chart could be 1-2, and the page number for the IPO chart described in the first block of the hierarchy chart could be 1-2 (see Figure 1). Let's add one more item to our index. That item is "module name," and is merely a short name (preferably 1 to 6 letters) that will be used to identify a given section of "code" when our program is written.



ABBREVIATIONS USED:

"HC" - Hierarchy Chart
"IPO" - Input, Process, Output Chart
"P" - Problem
"SP" - Subproblem

RDA126-10

Figure 1. Hierarchy Chart

Try and select a module name that will have some meaning. One way of doing this is to extract key letters from the description of the process performed by the module. The example below shows the completed index for the "GO TDY" problem, as far as it has been solved. Study the module names to see how they were derived.

NOTE: This index refers to the hierarchy chart on page 12.

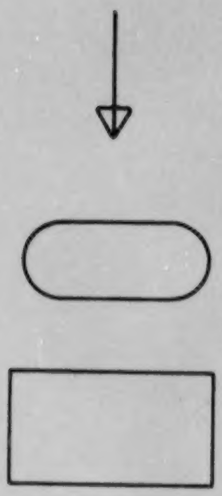
Key	Description	Page #	Module Name
1	Go TDY	1-3	GOTDY
2	Gather needed information	2	GNINFO
2A	Find knowledgeable person	2A	FKPRSN
2B	Get date, time, and location	2B	GDTLOC
2C	Get mode of travel	2C	GTMODE
3	Plan automobile trip	3-3	PCTRIP
4	Travel by car	4-3	TBYCAR
5	Plan plane trip	5	PPTRIP
6	Travel by plane	6	TBYPLN

If we examine the above index, knowing our page numbering system, it is rather obvious that key items 1, 3, and 4 have separate hierarchy charts and indexes. Note also that the index always specifies the number and IPO chart. Be consistent! You are right. We haven't developed the hierarchy charts for key items 3 and 4. That's your job.

Using this type of key and page numbering system will allow our HIPO package to grow without destroying our index system each time something new is added. This statement implies that we expect our HIPO package to grow as we solve the problem. It will, won't it? This should probably tell us that it would be a good idea to keep our developing HIPO "documentation" package in a looseleaf notebook so we can add pages as they are needed.

FLOWCHARTING THE PROBLEM SOLUTION

After we have completed the subdividing process and have a complete HIPO documentation package at our disposal, we have available a set of specifications which tell us what has to be done and to some extent when it has to be done. We now need some means of describing how something is to be done and when it should be done. This is called the "program logic" or "control logic" and may be shown using something called a flowchart. A flowchart is a picture that shows all functions or processes that must be performed and all decisions that must be made. Before we get into the process of showing how to draw a flowchart, we would like to define some figures we will be using.



RDA126-11

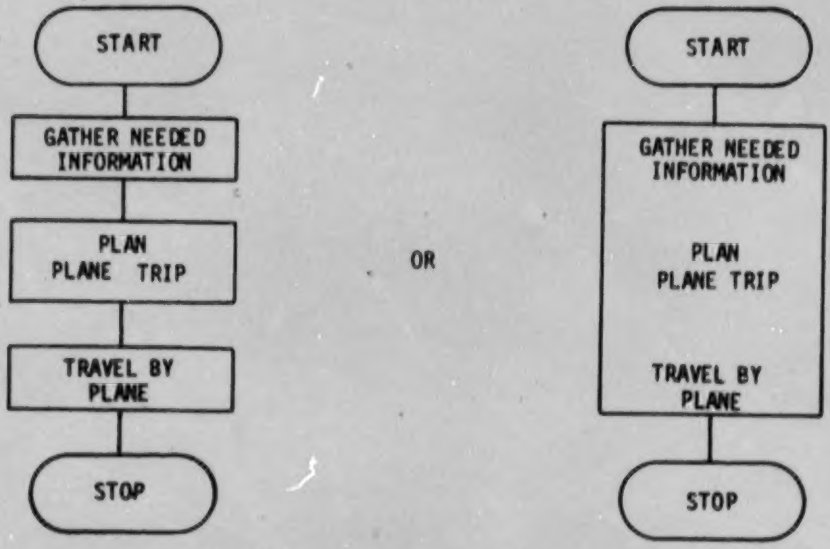
A line shows "control flow" or which operation is to be performed next. If an arrowhead is present, the arrow points to the next operation. If no arrowhead is present, the next operation is either "down" or to the "right."

Terminal Symbol. Either start control flow here or, when control flow reaches here, the job is done.

A process, or function, box. Only one control flow line may enter this box, and only one control flow line may exit this box. A text written inside the box describes what function or process must be performed.

The Sequence Primitive

A very simple type of flowchart is called a sequence flowchart. It pictures a series of processes that must be performed, one after another, in order. Let us assume for a moment that we are going TDY to Washington (again?), and that we will be traveling by plane. Our flowchart would look like this:

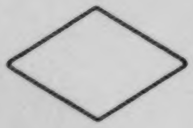


RDA126-12

Either of the preceding examples is correct. Any time we have a sequence of functions or processes to perform, they may be shown with each process listed in a separate function box, or all processes listed in the same function box. Any combination in between those two extremes is also acceptable. In other words, any sequence of functions is itself a function. A sequence of functions is also called a "control logic primitive." A control logic primitive is merely a predefined arrangement of flowchart symbols.

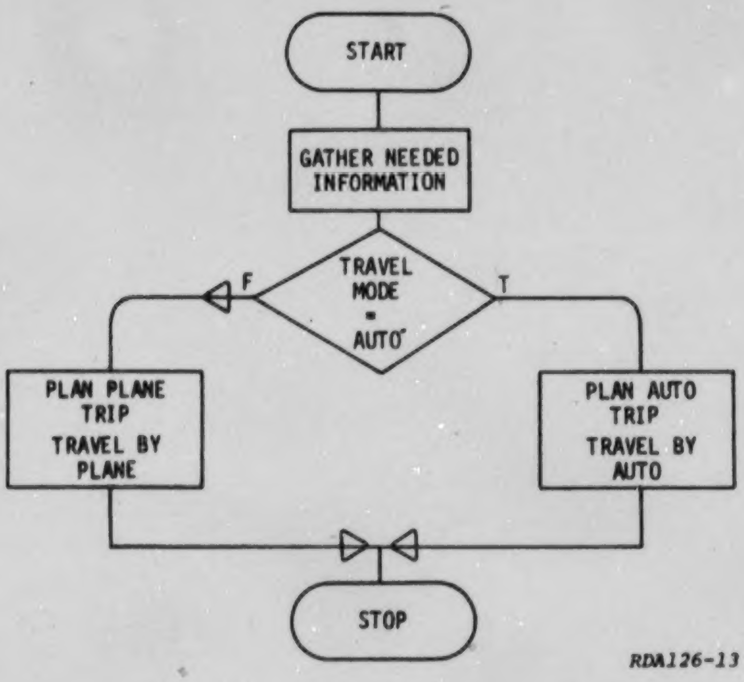
The IF...THEN...ELSE... Primitive

There are many places where a sequence of functions can be used in a flowchart, but very few problems where the entire solution method can be represented as a sequence of functions. Other types of control logic primitives will be needed. These other control logic primitives make use of a test for some condition, and then performance of a function based on the result of the test. Each control logic primitive is itself a function. The symbol used to indicate a test is a diamond, and is shown below:



Test of IF symbol. Only one control flow line may enter this symbol, but two or more must exit. The condition being tested for is stated inside the diamond.

The first control logic primitive we will discuss is known in programming circles as the "IF...THEN...ELSE..." primitive. The name of this function may sound strange at first, but it is a very descriptive term. IF (a condition is true) THEN (perform some process) ELSE, or otherwise (perform some other process). As a pictorial example, we will go back to our TDY problem.



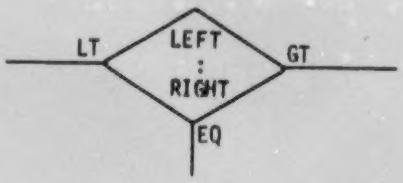
RDA126-13

Did you notice the letters "T" and "F" beside the decision box? They stand for "True" or "False" and indicate which path to take based on the result of the test. If Travel Mode was equal to Auto, we would follow the control flow line out the "T" side of the decision box and then Plan Auto Trip, etc. The letters "Y" for yes and "N" for no may be substituted for the T and F.

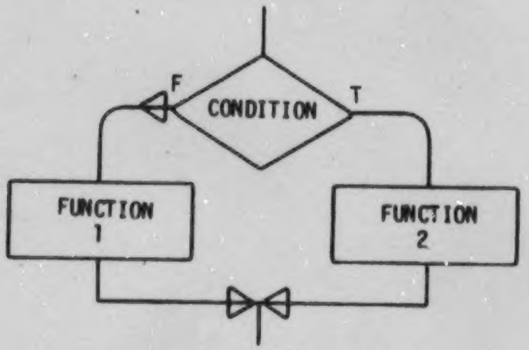
It may be helpful at this point to define some shorthand notation to use to describe the conditions we wish to test inside an IF diamond. We will use these symbols throughout the remainder of the course with no further explanation. In the descriptions below, the "left term" is defined to be the value to the left of the notation. Similarly, the "right term" is the value to the right of the notation, e.g., Left EQ Right, or X = 27, or L GT 4, etc.

NOTATION	DESCRIPTION
EQ or =	Is the left term equal to the right term?
NE or ≠	Is the left term not equal to the right term?
LT or <	Is the left term less than the right term?
GT or >	Is the left term greater than the right term?
LE or ≤	Is the left term less than or equal to the right term?
GE or ≥	Is the left term greater than or equal to the right term?

Whenever any of the above terms are used in a test, the exits from the test diamond must be labeled either T and F or Y and N. One other notation is used. It is a colon (:) and is used to compare the left term to the right term. The exits from a test diamond which uses a colon must be labeled with the notation described above to indicate the results of the comparison as shown below:

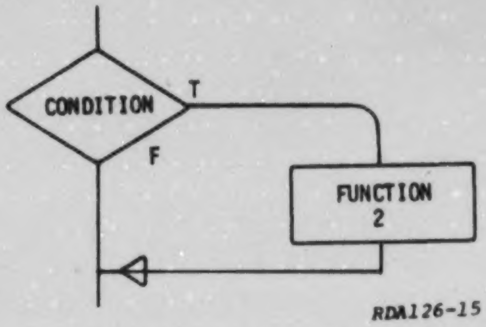


In general, the IF...THEN...ELSE primitive may be represented as follows:



RDA126-14

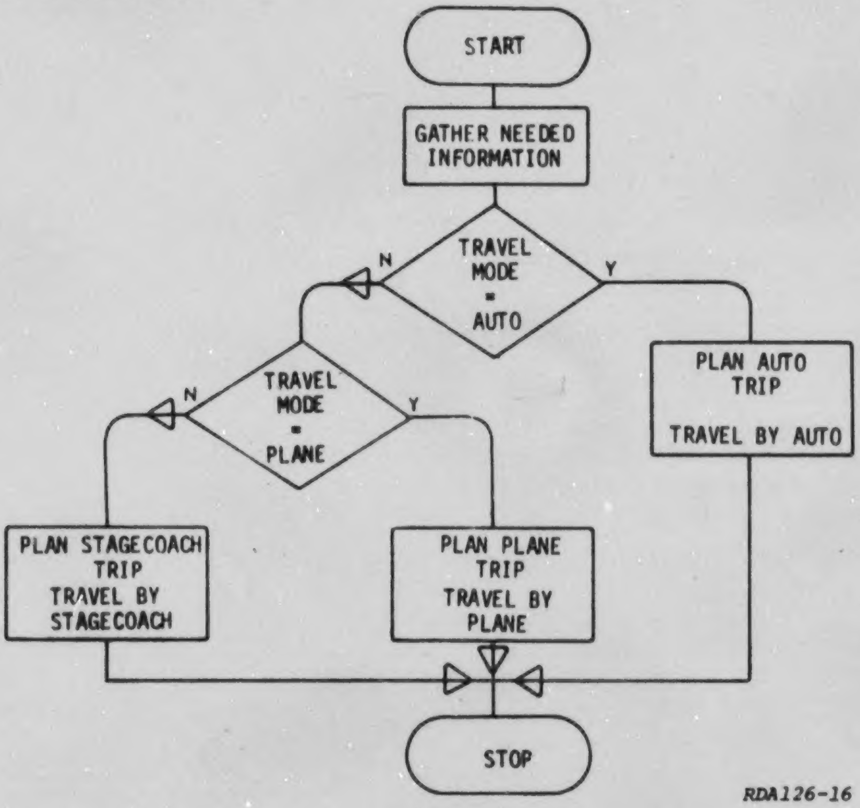
Function 1 and Function 2 are any processes, or control logic primitives. In addition, either Function 1 or Function 2 could be what is called a "null function." The null function is merely the absence of a function, as shown below:



RDA126-15

In the above example, Function 1 is a null function and is not shown. Why can only one of the functions in an IF...THEN...ELSE primitive be a null function?

We said earlier that control logic primitives were themselves functions. This implies that we can substitute an entire control logic primitive for any function box anywhere in a flowchart. As an example, let's assume that there are three modes of travel for our TDY problem--auto, plane, and stagecoach. If the mode of travel is not auto, we would exit the decision box of the previous example on the false "leg." At that point we would now have to determine if the mode of travel was by plane or stagecoach as shown below:

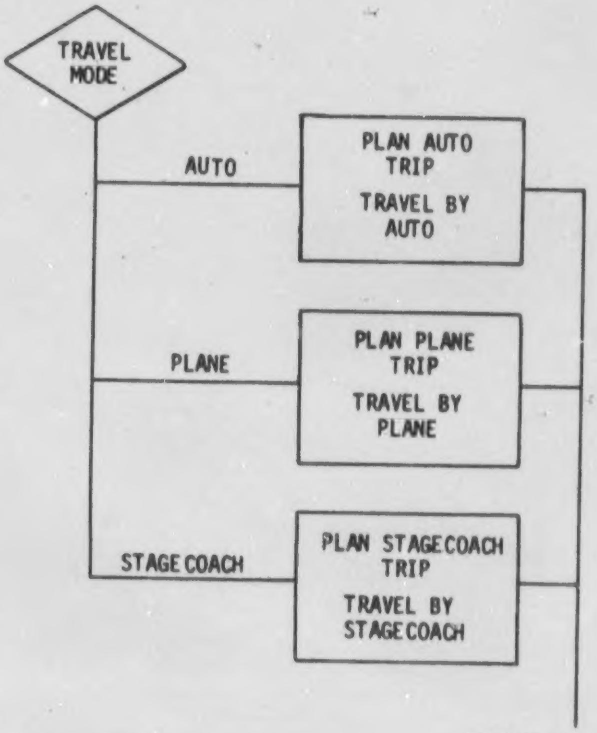


RDA126-16

When we substitute a control logic primitive for a function box inside another control logic primitive, it is called "nesting." In the preceding sample we have nested an IF...THEN...ELSE primitive inside another IF...THEN...ELSE primitive. This is one method of nesting; however, nesting can be carried to any level necessary to solve the problem.

The Case or Switch Primitive

If your particular problem requires you to nest several levels of IF...THEN...ELSE primitives, you may want to use the "case" or switch primitive. The case primitive uses the test diamond with one input control line, but the diamond has three or more output control lines. The previous example would look like the following if the case primitive were used:



RDA126-17

Notice that the case primitive has one control flow line entry and one control flow line exit and, as such, it also is a proper function. It may have as many function boxes as necessary to show the problem solution method.

To help clarify what we have just said, let's see how we might go about drawing a flowchart of a little more complicated problem. Our problem is to submit a loan application to the institution with the most favorable interest rate to finance a car we want to purchase. The IPO chart for this subproblem looks like the one on the following page.

SUBMIT APPLICATION FOR AUTO FINANCING

INPUT	PROCESS	OUTPUT
Year of car we wish to purchase (CAR)	Determine most favorable interest rate	Loan application completed
Number of payments we wish to make (NOPAY)	Submit loan application	
Interest Rate Chart (see below)		

RDA126-18

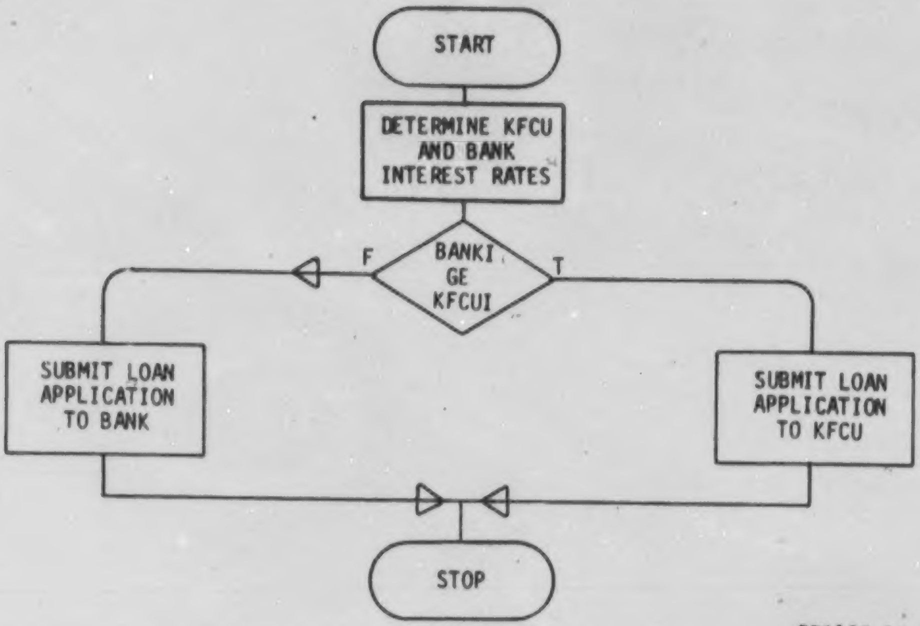
INTEREST RATE CHART

Type of Car	Length of Loan	KFCU Interest Rate	BANK Interest Rate
New	36 months	10.5	12.0
New	24 months	8.0	8.0
New	12 months	5.0	4.0
Used	24 months	10.0	12.0
Used	18 months	8.7	9.0
Used	12 months	6.5	6.0

KFCU refers to the Keesler Federal Credit Union of which you are a part and, thereby, part owner. If the interest rates between the bank and KFCU are the same, you will use the KFCU. In the flowchart, when we refer to the word "car," we are referring to the type of car (new or used); "NOPAY" will be the number of payments we wish to make, and "KFCUI" and "BANKI" will refer to the interest rates charged by the KFCU and the bank.

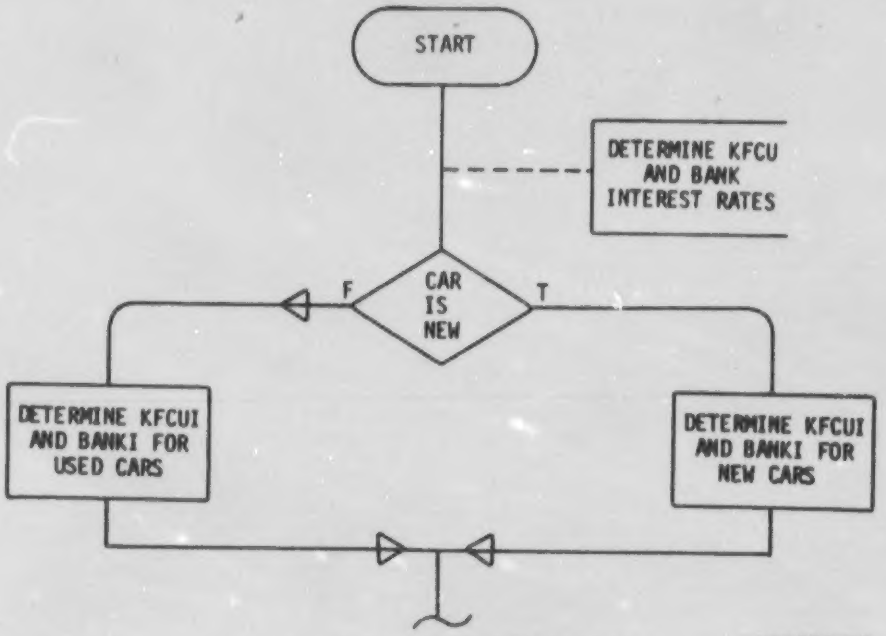
We will tackle the problem of drawing a flowchart in much the same way as we developed our IPO charts--one step at a time. We will draw several flowcharts to describe different parts of the problem; then we will draw a finished product which will show the whole solution in one flowchart. We can use scratch paper for our development of the flowchart. The final form, however, should be on paper that can be put in our HIPO documentation package.

Enough of details. Let's press on! Remember that we have to describe how we are going to perform the processes listed in the IPO chart. As we look at the processes, we can see that "Submit Loan Application" must be performed last. Look at the first process. The words "most favorable" jump out at you and say DECISION. If we take the decision out of that process, we are left with "determine interest rate." In order to make a decision, we need two or more pieces of information and, of course, we have the KFCU interest rate and the BANK interest rate. Our first attempt at a flowchart would look like the one shown at the top of the next page.


START
DETERMINE KFCU AND BANK INTEREST RATES
F
BANKI GE KFCUI
T
SUBMIT LOAN APPLICATION TO BANK
SUBMIT LOAN APPLICATION TO KFCU
STOP

RDA126-19

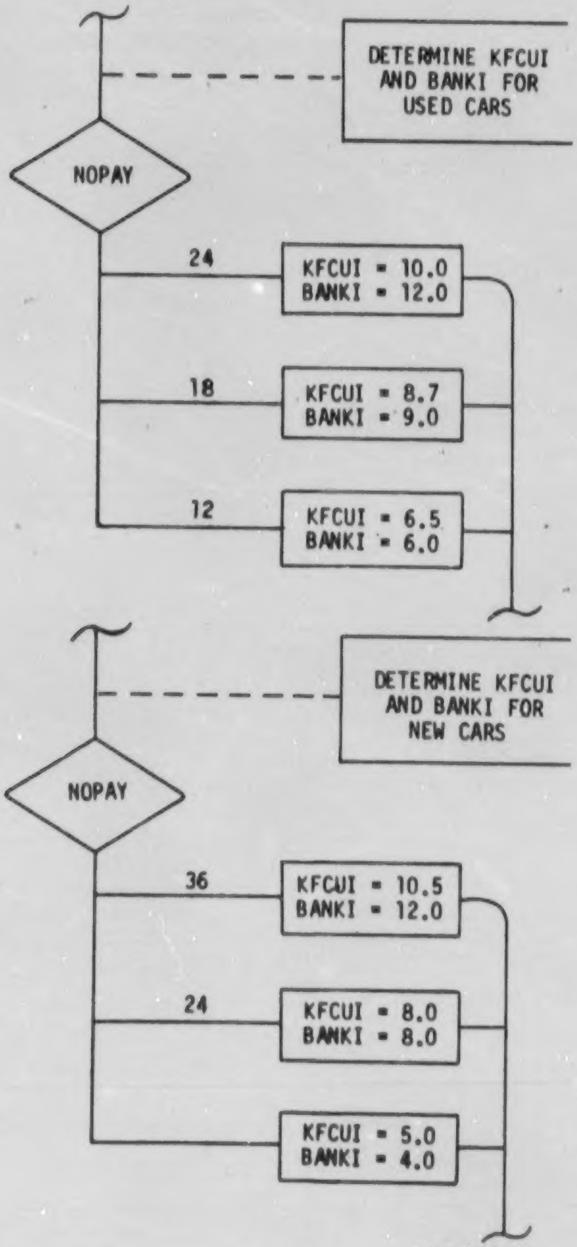
The function box that says "Determine KFCU and BANK Interest Rates" doesn't tell us very much, does it? OK! How do we determine what the interest rates are? We have to know if it is a new or used car, right? Put it down on paper.


START
DETERMINE KFCU AND BANK INTEREST RATES
F
CAR IS NEW
T
DETERMINE KFCUI AND BANKI FOR USED CARS
DETERMINE KFCUI AND BANKI FOR NEW CARS

RDA126-20

That simplified things somewhat— except for the two new items that showed up from nowhere. The open-ended box connected to the flowchart with a broken line serves to describe or "annotate" what this flowchart does. The curved line at the end just serves to remind us that this is only part of a flowchart.

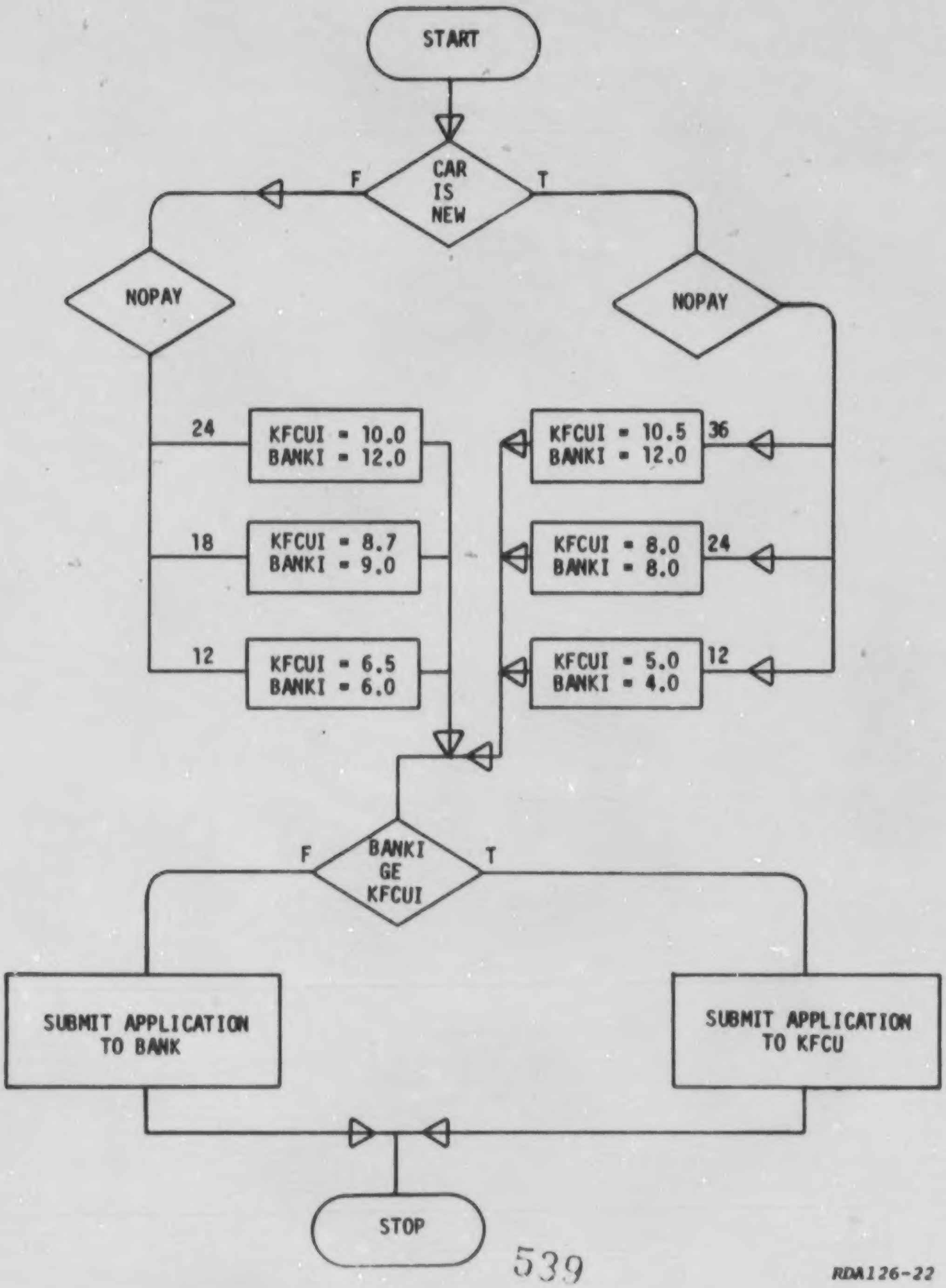
Back to the problem at hand. What else do we need to determine the interest rates? Of course! The length of the loan, or NOPAY.



RDA126-21

544

OK, we are almost done. All we have to do now is to put all the pieces together. See the flowchart below.



539

RDA126-22

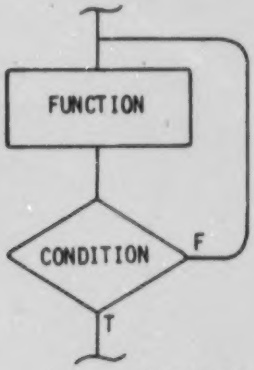
There! And you thought this was going to be hard. Remember the old saying that "even the longest journey begins with one small step"? Make use of that insight into problem solving. Draw your flowcharts a little at a time and you will be surprised how easily everything will usually fall into place.

You may be asking yourself, "Why didn't we continue breaking down the problem using an IPO chart?" Well, there is no convenient way we can show any decisions in an IPO chart. In fact, when a specific problem has been subdivided to the point where any further division would force you to show a decision that must be made, then it is time to stop the subdividing process.

The remaining control logic primitives we will discuss have one thing in common. They visually represent a repetitive algorithm, or a "logic loop." A logic loop is really the repetitive performance of one or more processes or functions. The number of times a loop is performed may vary from zero up.

The DO UNTIL Primitive

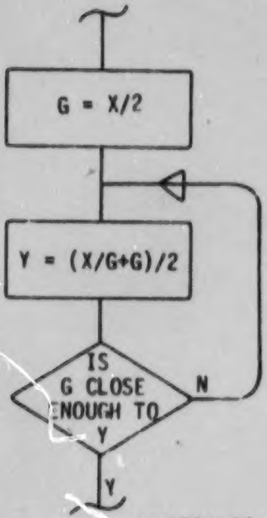
The first repetitive primitive we will examine is called the "DO UNTIL" primitive, and stands for the phrase "DO (perform a function) UNTIL (some condition is true)." When we perform a DO UNTIL loop, we enter the loop; perform a function; test a condition; IF the condition is true, we exit the loop; otherwise (ELSE) we go back and perform the function again.



RDA126-23

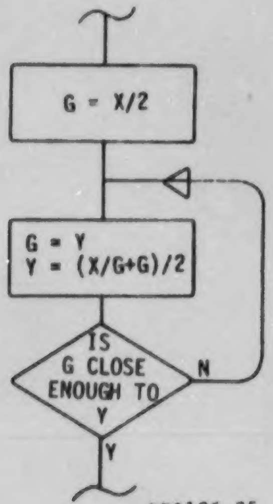
Let's look at the example we discussed back in the section on "The Scientific Trial and Error Method." We said to "Take the square root of a number using a 4 function calculator." The value we want is Y which is the $\sqrt{X}$. Remember, we said to start out with some number (G) which is equal to X/2. Next, perform the calculation Y = (X/G + G)/2. Then we test to see if Y is sufficiently close to G. If not, we set G = Y and perform the calculation again.

This problem might take some thought. First, we have to set G = X/2 outside the loop. (Why?) Then we perform our calculation and then the test. See the example at the top of the next page.



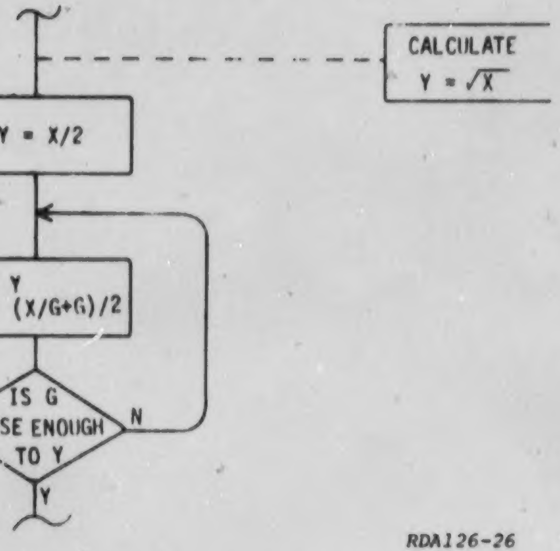
RDA126-24

Wait a minute! Will that work? NO! We didn't set G = Y after the test. Let's try adding another equation just before our calculation.



RDA126-25

Is out flowchart correct now? NO. The first thing we do is set G = X/2, but then we turn right around and set G = Y. What does Y equal? We haven't said, so we don't know. It could be anything--even zero. Can you divide X by zero? So, we still have a problem. How can we solve it? We could set Y = X/2 before we get into the loop, couldn't we? Good.



RDA126-26

That looks pretty good. The letter G starts out representing the right value. The equation Y = (X/G+G)/2 came out of a book on mathematics, so we will assume that it is correct. Therefore, if the values start out correct, and the calculations are correct, the answer must be correct. Two rights done't make a wrong???? or is that the other way around?

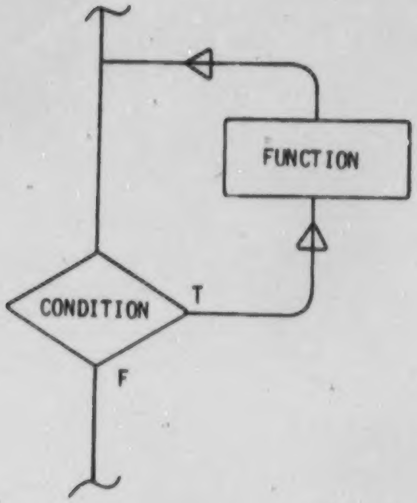
The function Y = X/2 was put in our flowchart to insure that "G" had a proper initial value when we started performing the loop. This process is called initializing the loop. It is just as important to insure that all loops are properly initialized as it is to insure that the functions performed within the loop are correct.

We used the Scientific Trial and Error method to develop the previous flowchart. That is perfectly acceptable. In time, as you become more familiar with loops, this method will become more scientific and less trial and error. Regardless of how you develop a flowchart, the finished product must be analyzed to doubly insure our flowchart tells us to do what we really want it to. It will usually be sufficient to show that our initial values are correct, the functions performed correctly the first time through the loop and the last time through the loop, and that our loop will eventually end.

You probably noticed that the function inside a DO UNTIL loop will always be performed at least once regardless of the state of the condition we will test for. This may not be the most advantageous way of performing every repetitive type of algorithm. To solve this, we will introduce a new primitive.

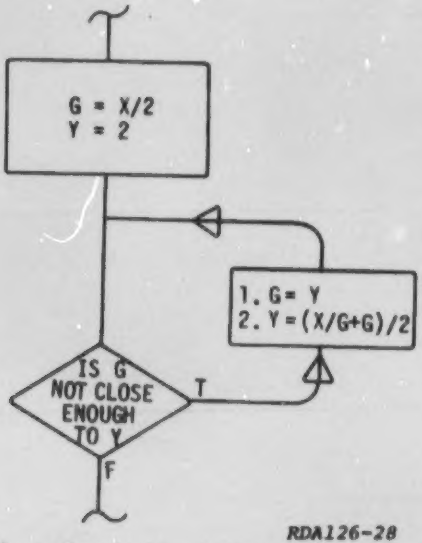
The DO WHILE Primitive

The DO (perform a function) WHILE (some condition is true) primitive checks loop terminating condition first. Then, if the condition is true, the loop function is performed and conditions checked again, etc. The general case of the DO WHILE primitive looks like the example at the top of the following page.



RDA126-27

Like other loops, the DO WHILE loop must be initialized. However, it is done differently than the DO UNTIL because the test comes at a different place. Our square root algorithm would look like the following if we used the DO WHILE primitive:

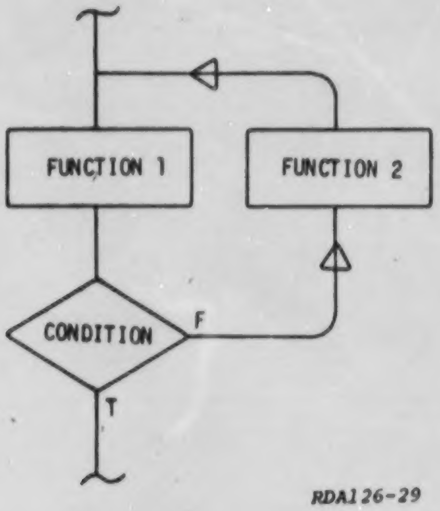


RDA126-28

There are two things worth noting in the above flowchart. First, look at the word "NOT" in the test condition statement. This effectively changed the result of the comparison so that a test that would have ordinarily given a True result now gives a False result, and vice versa. Second, the loop function has the control line flow passing through it from bottom to top. To avoid any confusion, we have numbered the functions in the order in which they are to be performed.

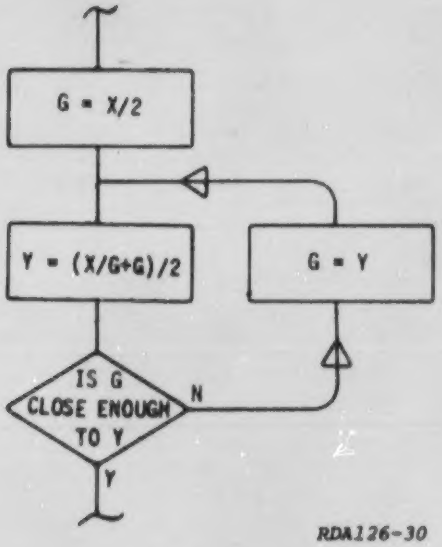
The LOOP EXIT IF Primitive

This is the last control logic primitive we will discuss. It is not really a unique primitive, but a combination of the DO UNTIL and DO WHILE primitives.



RDA126-29

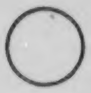
Our square root algorithm would look like the following if the LOOP EXIT IF primitive were used.



RDA126-30

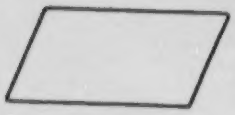
Miscellaneous Flowchart Symbols

Any problem solution can be flowcharted using only nested and sequential combinations of the control logic primitives we have previously discussed. As you are building a flowchart using these primitives, some of the following symbols may be of use.



The connector symbol is used in place of a control flow line to show control flow that passes from one page to another, or where a line would add confusion to the flowchart. Connectors are used in sets of two or more when used for this purpose. A label is given to each set of connectors, usually a letter and one or more numbers, but any logical label system will do. One connector (the starting connector) will have a control flow line entering it, while its pair (the terminating connector) will have a control flow line leaving it. Any terminating connector may be paired with more than one starting connector; however, all starting connectors must contain the same label as their associated terminating label.

The predefined process symbol can be used to indicate a process, function, or subproblem solution that is described by another flowchart. The text inside this symbol refers to the "MODULE NAME" of the process or subproblem to be performed. The module name is extracted from the verbal index to the hierarchy chart and is discussed on page 13. A predefined process is generally called a "subroutine" in programming circles.



The input-output symbol is used to show a process which makes available information for processing (input) or which records the results of the processed input information (output).

RDA126-31

General Guidelines for Flowchart Preparation

Always draw flowcharts of the highest level module first. In general, a module, or subproblem, should not be flowcharted until all other modules that refer to that subproblem as a predefined process have been flowcharted. You can hardly go wrong if you draw the flowchart of the highest level module (the executive) first. Then flowchart all modules on the second level of the hierarchy chart, then the third level, etc. This "Top Down" method is highly desirable, but it may not always be possible to flowchart all modules on one level before flowcharting some modules on a lower level. You may, for example, select one portion (or branch) of a hierarchy chart and flowchart all modules in that branch before going on to another branch, providing you use the top down method within that branch.

After you have drawn a flowchart of a module, you must analyze that flowchart. Every effort must be made to insure that the flowchart shows exactly what we want. If a flowchart contains some "logic error," any computer program written from that flowchart will contain the same logic error, and our program may do weird things (called a "bug"). While it is possible to remove logic errors from programs after they have been written, it is far better never to write a "bug" into a program. It has been reasonably stated that "If debugging is the art of removing bugs from computer programs, then writing computer programs must involve putting bugs into them."

After a module has been flowcharted, and that flowchart has been scrupulously analyzed, it is filed in the HIPO documentation package immediately behind the Input-Process-Output chart for that module. The size of any particular module has a lot to do with how effectively it can be analyzed. Each module should be considered in its own right, but generally speaking:

1. If a module involves less than four or five functions and tests, maybe it is too small to be a separate module. If this is the case, those processes performed by that module should be included in the next higher module in the problem hierarchy.

2. If a module involves many more than 40 functions and tests, perhaps our module is too complex for effective analysis. In this case, you should consider breaking the large module into two or more smaller modules that together will perform the same functions as the original module.

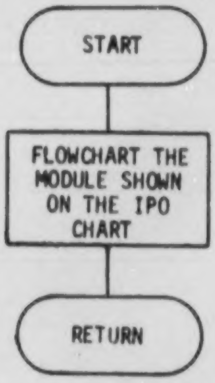
FLOWCHART DEVELOPMENT

We now have all the tools necessary to develop a flowchart from an IPO chart. Let us see if we can bring it all together and draw a flowchart which we can use later as a guide for developing any flowchart. We will assume that the problem definition phase has been completed and that we have an IPO chart at our disposal. The IPO chart would look like this.

DRAW A FLOWCHART

INPUT	PROCESS	OUTPUT
IPO Chart	Flowchart the module shown on the IPO Chart	Flowchart of the module logic

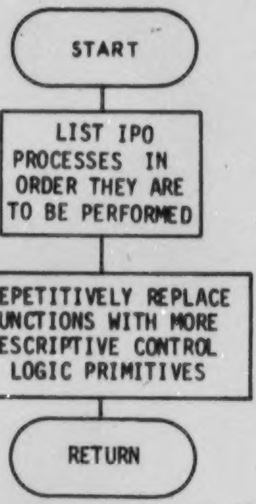
To begin with, we start and end the flowchart with the terminal symbols. Between the terminal symbols, we insert the processes listed in the IPO chart, using a simple sequence of functions.



RDA126-32

NOTE: The word "RETURN" has been used instead of STOP in the terminating symbol. This is used in a submodule or subroutine to indicate that the function performed by this module is complete. The control flow should return, or go back, to the module that originally referred to this module as a predefined process (calling module).

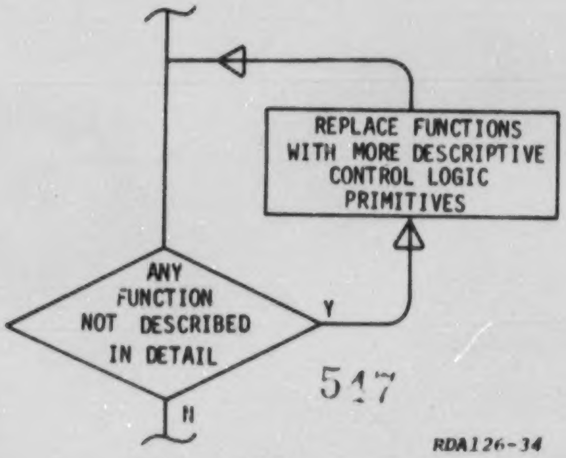
Now, if memory serves us correctly, we have to repetitively replace the functions of our flowchart with more descriptive control logic primitives.


START
LIST IPO PROCESSES IN ORDER THEY ARE TO BE PERFORMED
REPETITIVELY REPLACE FUNCTIONS WITH MORE DESCRIPTIVE CONTROL LOGIC PRIMITIVES
RETURN

RDA126-33

The function "List IPO processes. . ." seems complete and descriptive. However, the second function is vague, to say the least, and definitely needs to be refined and listed in more descriptive terms. Before we go on with the refining process, we must consider the new functions together and insure they describe the single original function. They do, in fact, describe the same process, so we can move on.

The word repetitively in the second function implies a process that should be performed in a loop. All three loop primitives should be considered and one selected that you feel will best serve your immediate needs. We will opt for the "DO WHILE" primitive, but the "DO UNTIL" primitive would probably also get the job done.

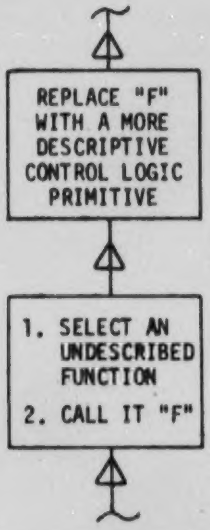

REPLACE FUNCTIONS WITH MORE DESCRIPTIVE CONTROL LOGIC PRIMITIVES
ANY FUNCTION NOT DESCRIBED IN DETAIL
Y
N

547

RDA126-34

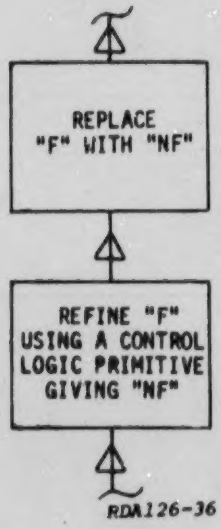
The substitution of the "DO WHILE" primitive has not changed since our original function description. The loop will eventually end. When? Everything seems to be in order, so let's press on.

It would be very difficult to attempt to replace all functions at once, so maybe our next step should be to select one function and replace that function with a more descriptive control logic primitive.


REPLACE "F" WITH A MORE DESCRIPTIVE CONTROL LOGIC PRIMITIVE
1. SELECT AN UNDESCRIBED FUNCTION
2. CALL IT "F"

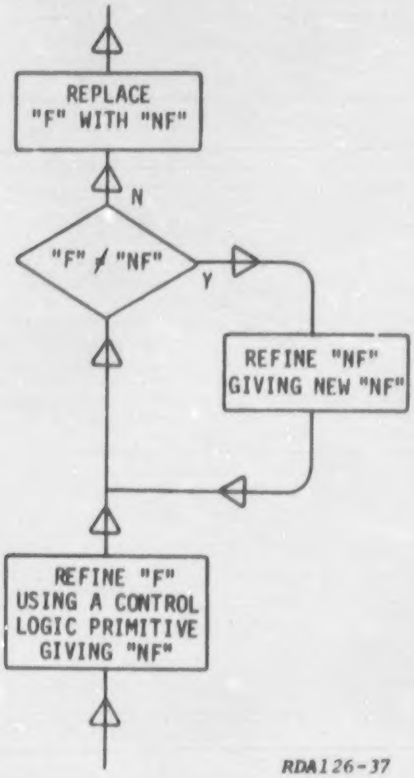
RDA126-35

The next step is to refine "F" using a control logic primitive and then replace "F" with the new refined function.



RDA126-36

If we analyze what we have, we see something has been forgotten. What is it? We made no provisions to insure that function "NF" is the same as function "F." We had better test NF = F before we replace F with NF. What do we want to do with NF if it is not equal to F? We probably want to refine NF and test again, right? That means we need to insert another "DO WHILE" primitive before we replace F with NF. We now have the following:

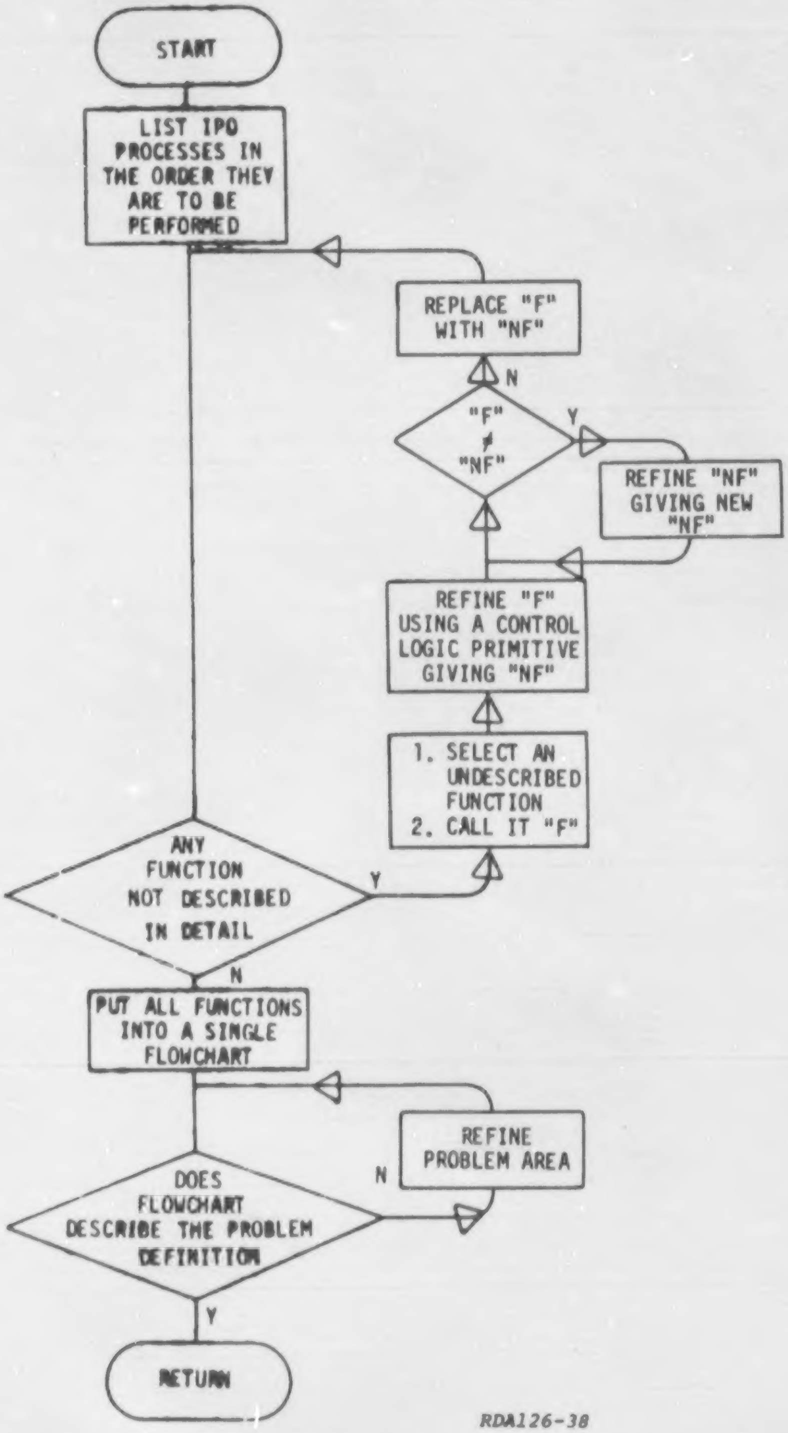

REPLACE "F" WITH "NF"
N
"F" ≠ "NF"
Y
REFINE "NF" GIVING NEW "NF"
REFINE "F" USING A CONTROL LOGIC PRIMITIVE GIVING "NF"

RDA126-37

We now have all functions described. The only thing left is to put all the parts together into one flowchart showing all functions. But wait! We haven't shown that step at all in our flowchart. When should that function be performed? After all functions have been described and before we return to the calling module.

After the flowchart has been completed, the entire flowchart must be analyzed to insure that the flowchart in fact describes the problem definition. If a problem is discovered, refine the problem area and check the flowchart again. This is the last process performed before returning to the calling module.

Our overall flowchart for this module will now look like the one below.


START
LIST IPO PROCESSES IN THE ORDER THEY ARE TO BE PERFORMED
REPLACE "F" WITH "NF"
N
"F" ≠ "NF"
Y
REFINE "NF" GIVING NEW "NF"
REFINE "F" USING A CONTROL LOGIC PRIMITIVE GIVING "NF"
1. SELECT AN UNDESCRIBED FUNCTION
2. CALL IT "F"
ANY FUNCTION NOT DESCRIBED IN DETAIL
Y
N
PUT ALL FUNCTIONS INTO A SINGLE FLOWCHART
REFINE PROBLEM AREA
N
DOES FLOWCHART DESCRIBE THE PROBLEM DEFINITION
Y
RETURN

RDA126-38

Arrays and Subscripting

We will now take what we have learned and applv it to some problems that are more closely related to computer programming. Obviously, we cannot show you flowcharts that describe every possible problem you may be given to solve. If we could do that, there wouldn't be any need for your job. What we can do is show you the thought process that goes into solving a few select computer-related problem types. You may be able to use the solution to these problem types from time to time. Generally speaking, however, you will have to develop your own solution method to a unique problem. With this in mind, it would behoove you to strive for an understanding of not only what we are doing, but HOW we go about developing a solution method. In this case, the result is not nearly as important as the method of arriving at that result.

Computer programmers talk about manipulating, or working with, "tables," "arrays," "memory locations," and other such abstract stuff. For our purposes, think of a memory location as a small box in which we can put information, or can get information out of. A "table" (not the four-legged kind you eat from) and an "array" refer basically to a group of memory locations, or boxes, placed next to each other.

An array of boxes placed side by side on a shelf could be given one name such as "TRAY." The name "TRAY" would refer to the entire group of boxes on the shelf. Each box could then be given a number, starting with 1 for the box farthest to the left, the number 2 would be given to the box immediately to the right of box number 1, box number 3 would be immediately to the right of box number 2, etc. To refer to the contents of a specific box, we would then enclose the box number inside parentheses ( ) immediately following the array name. For example, if we wanted to refer to the contents of box number 4 in an array called "TRAY," we would write TRAY(4).

A common real life array you are probably familiar with is a tray that holds 35mm color slides to be used with a slide projector. A slide tray has anywhere from 20 to 120 different slots in which you can place a slide. Each slot is numbered starting from 1. If you want to get the slide that is in the 20th position of the tray, you write SLIDE(20). SLIDE is the name of the array.

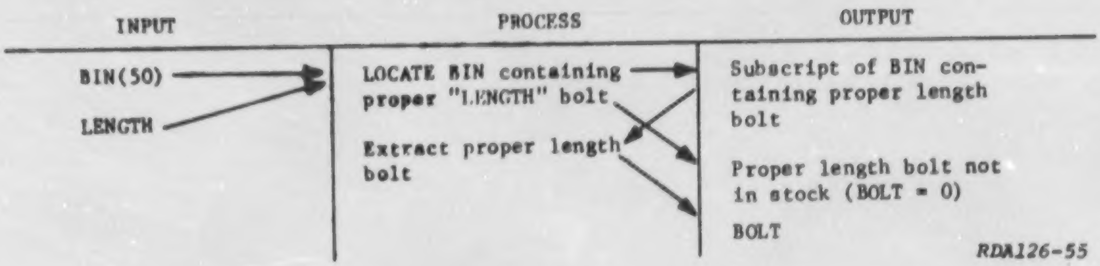
In programming terms, what we have just described is called "subscripting." The name of the array is called the "subscripted variable," and the number inside the parenthesis is called the subscript. In our example above, SLIDE is a subscripted variable, and 20 is the subscript.

There is no rule that says that a subscript has to be a number. It could be a letter, or even a name, that represents a number (called a variable). For example, if we set I = 20 and then write SLIDE(I), we would be referring to the same slide as if we wrote SLIDE(20). Using a variable as a subscript has a big advantage when we want to refer to different elements or boxes in an array while we are performing a loop. We can change the value that a variable represents, but we can't change the value a number represents. If this concept seems a little hazy to you now, don't worry too much about it. As we go through the examples in the sections that follow, it should become clearer.

Sequential Search Algorithm

Assume for a moment that you are a stock clerk. In your stock room you have a shelf with 50 bins on it. Each bin contains a different length 1/4-inch bolt. If someone comes to you wanting a certain length bolt, how do you find the bin that contains the right bolt? The solution method to this problem is what a search algorithm describes. The IPO chart for a search module would probably look like the chart at the top of the next page.

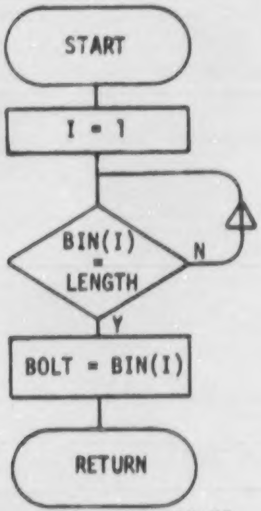
LOCATE AND EXTRACT PROPER LENGTH BOLT


INPUT
PROCESS
OUTPUT
BIN(50)
LENGTH
LOCATE BIN containing proper "LENGTH" bolt
Extract proper length bolt
Subscript of BIN containing proper length bolt
Proper length bolt not in stock (BOLT = 0)
BOLT

RDA126-55

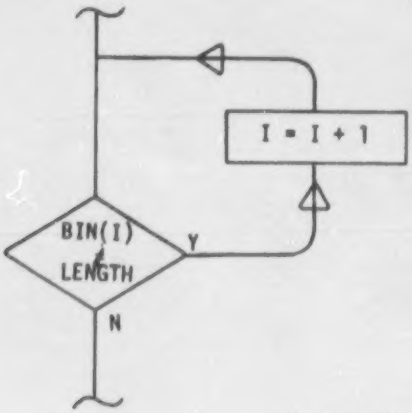
The input column tells us we have an array of 50 bins, and LENGTH is a variable which represents the length of the bolt we wish to find. When we perform the first process listed, one of two things will happen. Either we will find the right length bolt, or we won't. If we find the bolt, we have no problem; however, if we don't have that length available, we need some way of telling that to the calling module. We can do that by returning a bolt of length zero (or no bolt) by using the algebraic equation BOLT = 0.

This looks like a problem we might solve using a loop and a subscript to check the different bins for LENGTH. Remember, we said we could use a variable as a subscript. We will use the variable I. We will do this the easy way and check BIN(1) first. If BIN(1) doesn't contain the proper length bolt, then we will check BIN(2), etc., until we have checked BIN(50). If we have checked BIN(50) and still haven't found the length bolt we want, we will assume we do not have the bolt in stock. However, we won't worry about this problem just yet. Since we want to check BIN(1) first, we will want to initialize our subscript to 1. Then, inside a loop we want to check to see if BIN(I) = LENGTH. When BIN(I) = LENGTH, we have found what we are looking for, so we want to branch out of the loop and set BOLT = BIN(I).



RDA126-39

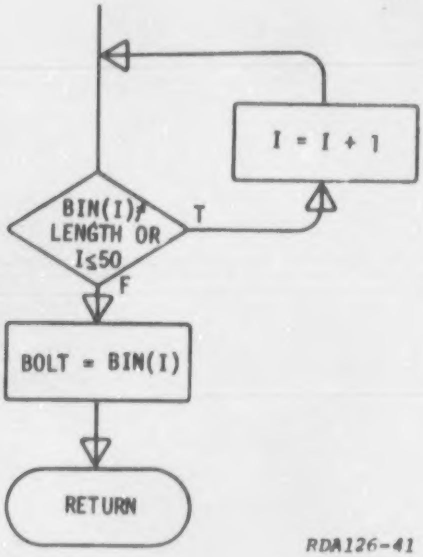
Now, go back and analyze what we have. Will it work? Why not? Will we ever check any bin except BIN(I)? No, we won't, because we never changed I. We need a function inside the loop that will add 1 to the variable I (called incrementing I) each time the loop is performed. What algebraic expressions will do that? Will the expressions J = I + 1 followed by I = J do the job? They sure will. We could simplify this function by just saying I = I + 1, which reads "let the new value for I equal the current value of I plus 1." Our loop now looks like this:



RDA126-40

If we take this refined loop, put it in our original flowchart, and analyze the result, we find that we will check to see if LENGTH is equal to BIN(1), then BIN(2), then BIN(3) . . . then BIN(50), then BIN(51), etc., until we find a bolt that is the right length. But, wait! How can we check BIN(51) if we only have 50 bins? We can't, so we must include some means of stopping the loop if we don't find the length bolt we are after.

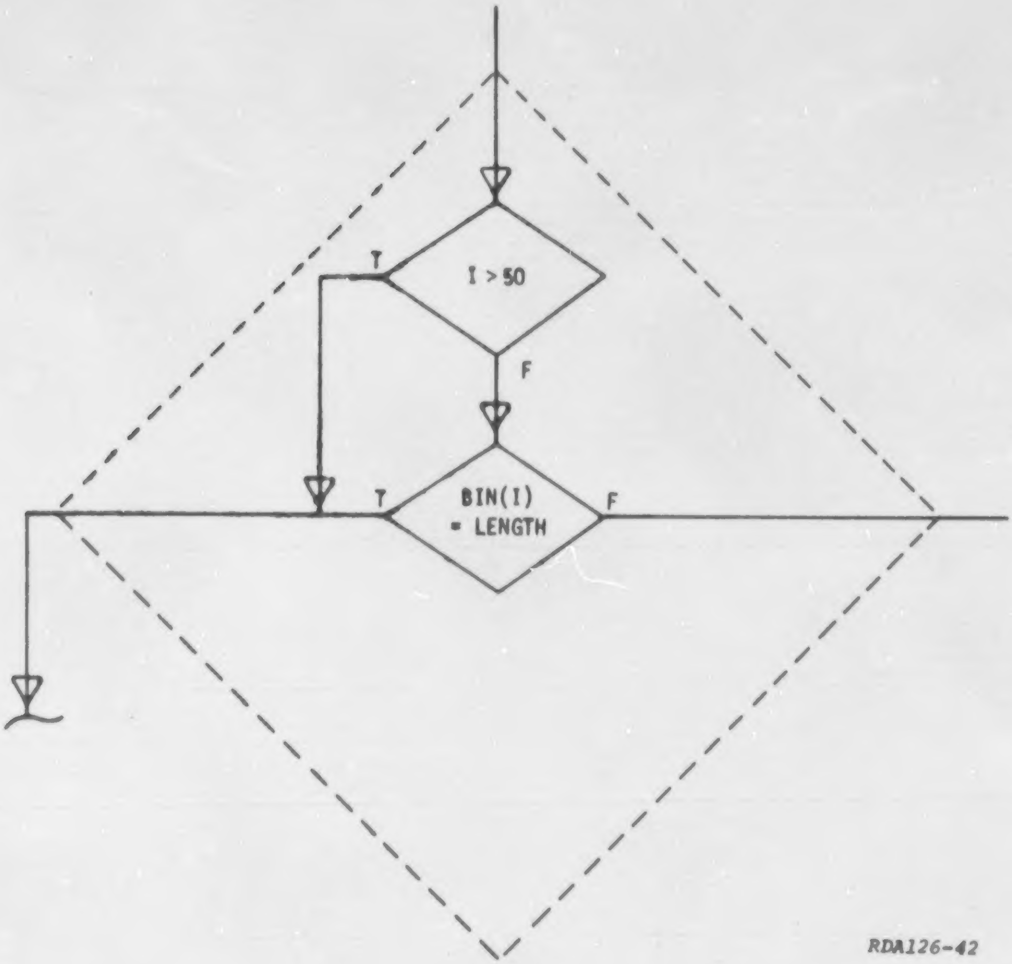
What is the only way we can tell that we do not have the right length bolt in stock? If the variable I ever gets to be 51, what has happened? We searched all 50 bins and didn't find the bolt. We could change our loop terminating condition so that we would exit the loop if either BIN(I) = LENGTH, or I > 50.



RDA126-41

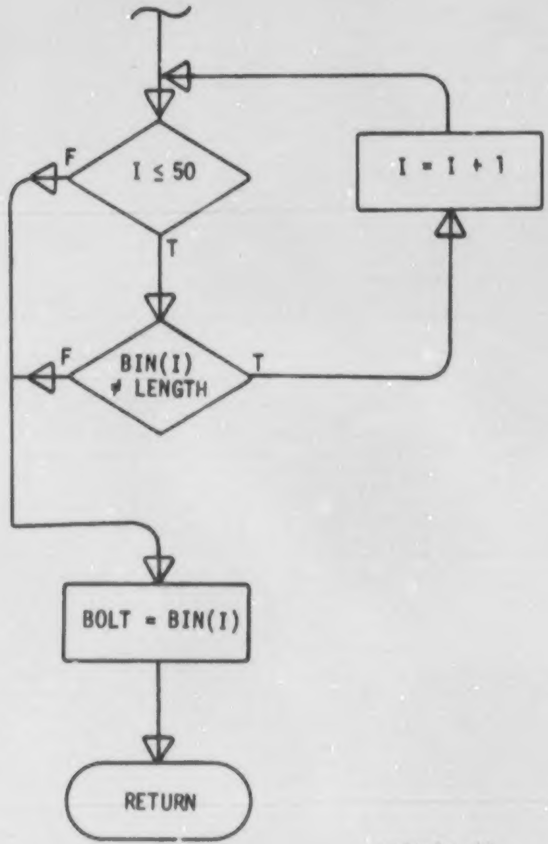
559

Sometimes it becomes very difficult to follow the logic if more than one condition is tested at a time. It would be simpler if we could show complex tests as a series of IF diamonds. You must be very careful when using a series of IF diamonds to insure that you do not destroy the structured format of your flowchart. There must be only one entrance to a sequence of IF diamonds, and two exits--one for True and one for False. No process or function boxes may be shown within the sequence of IF diamonds. In short, you must be able to draw a large diamond around your sequence of IF diamonds, and have one entrance and two exits as shown in the following example.



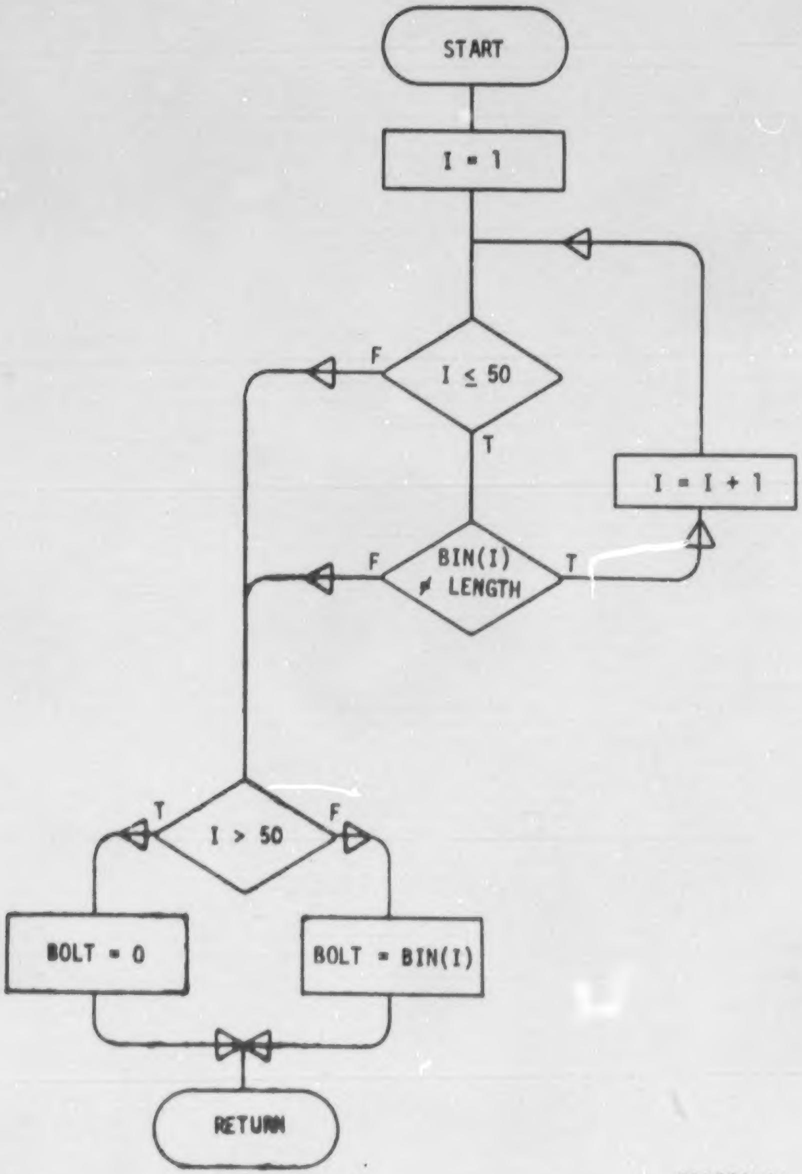
RDA126-42

We may now take this concept and use it for our own purposes in our search for the proper length bolt. We now have two conditions that could cause us to exit our loop: One for the normal condition of BIN(I) = LENGTH and the other exit for the abnormal condition of I > 50.



RDA126-43

Is the above flowchart segment correct? Do we want to set BOLT = BIN(I) if I > 50? No. We need to add another test outside the loop to see if we took the normal or abnormal exit. How can we find out which exit was taken? If I > 50, we must have taken the abnormal exit. If we take the abnormal exit, we want to set BOLT = 0. The complete flowchart now looks like the one on the following page.



RDA126-44

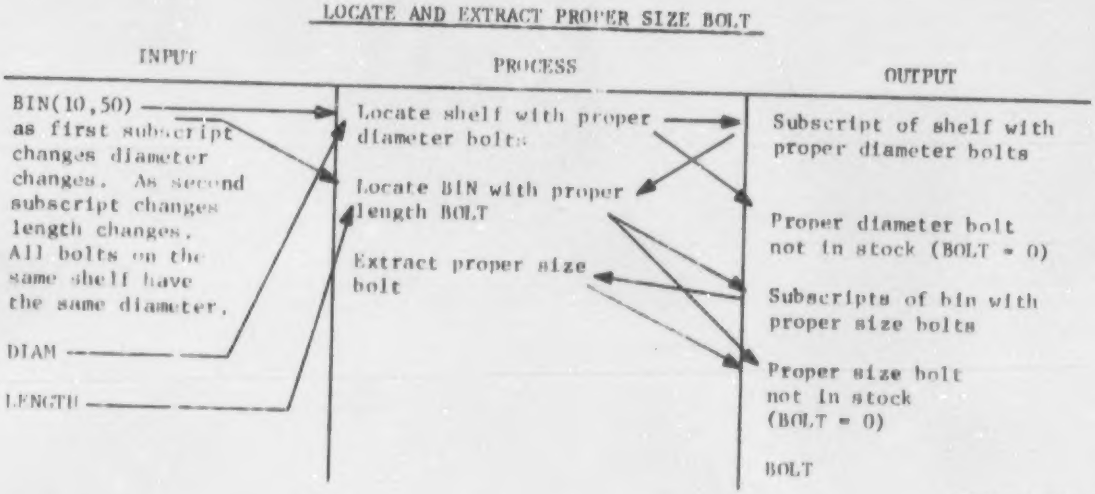
When we go back and analyze the completed flowchart, we see that the loop starts out correctly (I = 1 the first time I is used as a subscript); that the loop will eventually end (before I = 51 is used as a subscript or when we have found the bolt we want); and after we exit the loop, we see we will set BOLT equal to the proper value (0 or BIN(I), depending on the value of I). That is exactly what we wanted it to do--so we can say our flowchart is correct.

We can now take this subscripting concept another step further. Assume you are back in your stock room but, instead of having one shelf with 50 different bins, each containing a different length bolt, you now have 10 different shelves, and each shelf has 50 different bins. The bins on each shelf contain all bolts of a specific diameter. We have 10 shelves, so we have bolts in 10 different diameters. We would indicate this situation by adding another subscript to our array BIN. BIN(10,50) would tell us we have a 10 by 50 array. To reference the first box on the second shelf, we would write BIN(2,1) because the first subscript used refers to the shelf number, while the second subscript refers to the specific bin on that shelf. If we reverse the order of the subscripts and write BIN(1,2), we would be referring to the second bin on the first shelf. As you can see, it is very important to keep straight which subscript is used for which purpose when more than one subscript is used to reference an item in an array.

The number of subscripts used to reference an item in an array is also referred to as the dimension of an array. For example, an array with one subscript is called a single-dimensional array, an array with two subscripts is called a double- or two-dimensional array, etc.

How would we go about searching for an item contained in a double-dimensional array? Let's go back to our 10 shelves with 50 bins per shelf and see. For this problem, our IPO chart would look like this:

LOCATE AND EXTRACT PROPER SIZE BOLT


INPUT
PROCESS
OUTPUT
BIN(10,50)
as first subscript changes diameter changes. As second subscript changes length changes. All bolts on the same shelf have the same diameter.
DIAM
LENGTH
Locate shelf with proper diameter bolts
Locate BIN with proper length BOLT
Extract proper size bolt
Subscript of shelf with proper diameter bolts
Proper diameter bolt not in stock (BOLT = 0)
Subscripts of bin with proper size bolts
Proper size bolt not in stock (BOLT = 0)
BOLT

RDA126-56

The IPO chart looks a little complicated, doesn't it? Never fear, if we tackle the problem one step at a time as we have in the past, everything will just seem to fall into place.

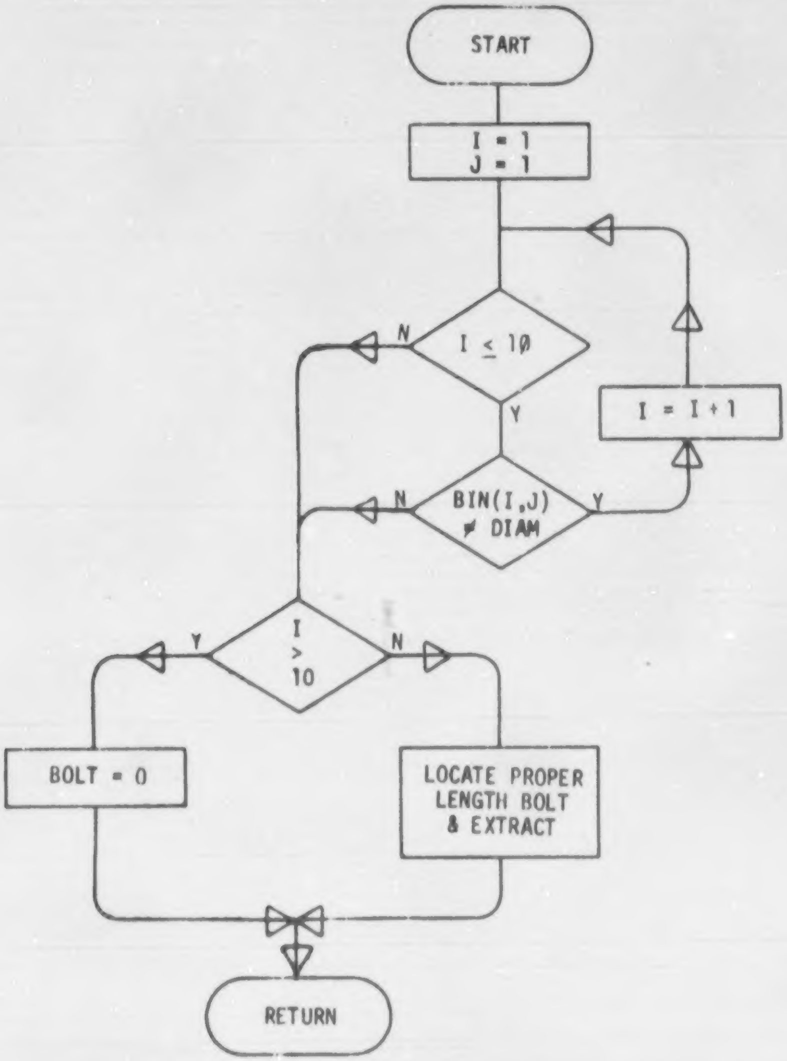
Did you notice the description of what happens as the subscripts in array BIN change? This type of information is very helpful to have in the input column of the IPO chart. This way we have all the information we need right at our fingertips.

The first thing we will have to do is initialize some variables to be used as subscripts. How many, and why? We will need two variables because array BIN has two subscripts. Let's use variable names I and J as our subscript. Variable I will be used to indicate which shelf we are on, and variable J will be used to indicate which bin on shelf I we are working with.

Do we want to find the right diameter or the right length bolt first? Well, we know that all bolts on a shelf have the same diameter (from the INPUT column of the IPO chart), but we do not know that the same bin number on different shelves contains the same length bolts. Therefore, it would be a good idea to locate the shelf that contains the proper diameter bolts first.

After we have exited the loop that performed the search for the proper diameter bolt, we must determine if we terminated our search normally or abnormally. How will we know? If we terminate abnormally, the variable I will be greater than what? Right! The I will be greater than 10.

If we terminate our search abnormally, we will set BOLT = 0 and exit this module. Otherwise, we will have to search shelf I for the proper length bolt.



You will note that the preceding flowchart is not complete. That portion of the flowchart that locates and extracts the proper length bolt from bin J on shelf I has not been expanded. That is your assignment for tomorrow.

After you have completed your flowcharting assignment, examine it carefully. What is the largest number of boxes you would ever have to check to find any given size bolt? You will note that you have to check less than 20 percent of the total number of bins to find any size bolt contained in your stock. That will save you a lot of time, won't it? Why don't you have to look in every bin? Could it be because the bolts were stored in order, by diameter? That is an important point to remember when you are designing a "data base" for a computer program. A program data base is really a lot of related, or even unrelated, information stored in a computer's memory. Remember, a computer's memory is made up of a whole lot of words, or little boxes, into which the computer can store information or from which it can extract information. Design your data base with some kind of order, and the computer will be able to locate any given piece of information much faster than if there was no order at all. How many bins would you have to look at to find a bolt if none of the bolts were stored in order? About 500.

Assume that each bin has two sections in it. The front section contains bolts with a hex head, and the back section contains bolts of the same length and diameter as the front section, but with a square head. Array BIN is now subscripted as follows: BIN(10,50,2). As an exercise in elementary non-trivia, you are to develop an IPO chart to describe the problem of locating a certain length, diameter, and head type. In addition, you are to draw a flowchart which will show all the steps necessary to solve the problem.

The sequential search algorithm we have just discussed is the simplest search technique of them all. It will work on information stored in a single dimension array, without any specific order. Its main drawback is that it requires more time to perform than some of the other more sophisticated techniques.

The Bubble Sort Algorithm

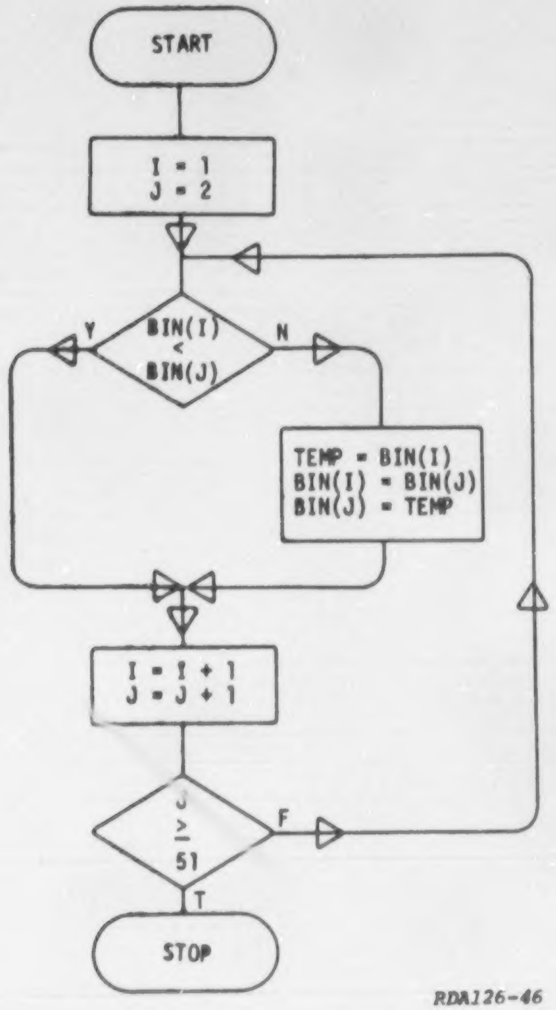
One of the more sophisticated search techniques, called the binary search, requires that information be stored in some ascending or descending order, i.e., alphabetical, numerical order with smallest first, numerical order with largest first, etc. You use a modified version of this technique every time you look a word up in a dictionary or look up someone's phone number. You certainly don't start looking on the first page of a dictionary for the word "ZYMURGY," do you? Of course not. A dictionary is in alphabetical order, so that technique works very well.

Assume that you have just become a stock clerk. You are replacing someone who was fired because he took a long time to find a certain size bolt. When you walk into the stock room, you notice a shelf that contains 50 bins. Each bin contains bolts of a certain size. All bolts on the shelf are the same diameter. However, the bins seem to be mixed up. There is no particular order, by length, to the way the bolts are stored on the shelf. You know that if the bolts were stored in order by length, you would be able to find any size bolt much faster than by doing a sequential search on a disorganized mess. Your problem, then, is to figure out how to sort the bolts and store them on the shelf in the order of their length.

The simplest, but by no means the fastest, way would be to use a bubble sort method. You look at the first bin and the second bin. Are the bolts in the second bin shorter than the bolts in the first bin? If they are, you simply exchange the bolts in the first bin with the bolts in the second bin. Then you move on and compare the bolts in the second bin with the bolts in the third bin. If the bolts in the third bin are

565

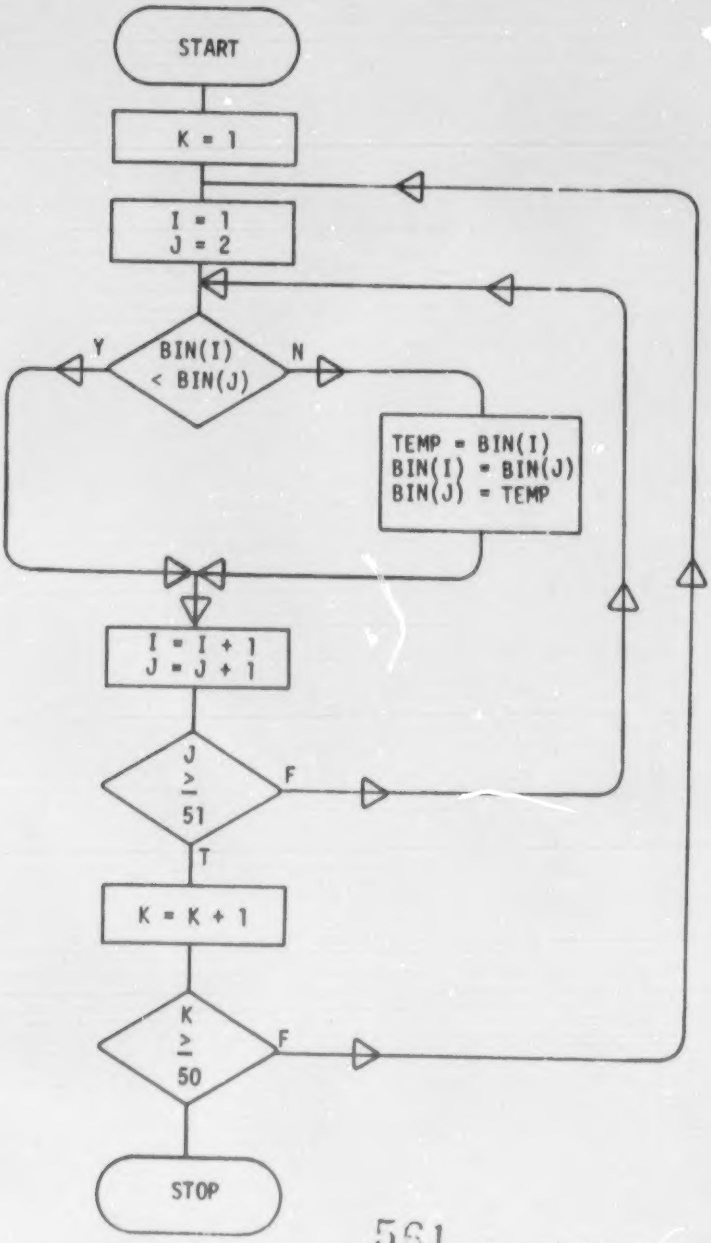
shorter than the bolts in the second bin, you exchange the contents of the two bins. You continue this process until you have compared the contents of the last bin with the contents of the next to the last bin and exchanged them if necessary. A flowchart of what we have described looks like the following:



RDA126-46

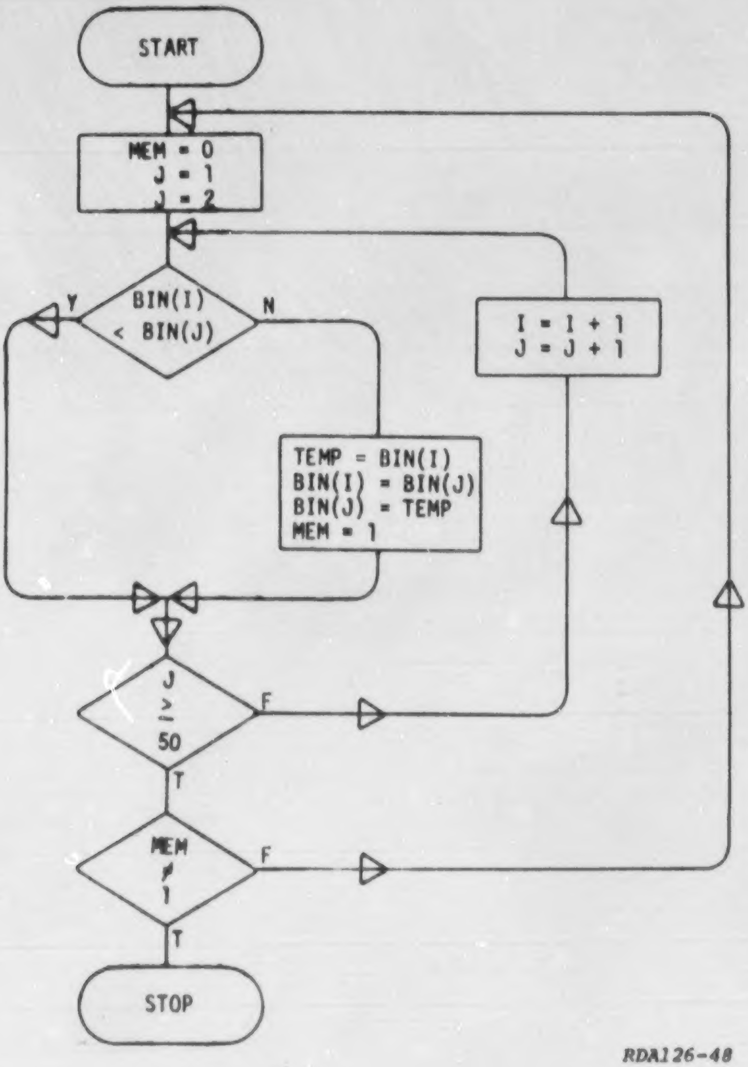
Why did the expression TEMP = BIN(I) appear in the function box of the IF...THEN... ELSE control logic primitive? What would happen if we dumped the bolts from BIN(2) into BIN(1) before we removed the bolts that were originally in BIN(1)? They would get mixed together and we would really have a mess. In a computer, if you put some information into a specific memory location, the original contents of that location will be destroyed, nevermore to be seen. We have to remove information we want to keep from one memory location before we put other information in that same location.

When we go back and examine the flowchart on the previous page, we see that it does just what we said it would do. But, will that operation put all of the bolts in proper order? If the shortest bolts on the shelf were in BIN(50) when we started, where would they be when we finished performing all the operations shown by the flowchart? The shortest bolt would be in BIN(49). We would have to perform all the operations shown on our flowchart 48 more times. We could set up an outer loop which would be performed a total of 49 times.



Will that do the trick? Yes it will. However, what if the bolts were all sorted after K = 13? We would continue to go through the loop until K = 50, and accomplish absolutely nothing. It would be far more efficient if we could think of a way to stop our outer loop soon after the BIN array was completely sorted.

How can we tell when the entire array has been sorted? If we perform the inner loop 50 times and never exchange bolts in two adjacent bins, the sort process has been completed. We could use a memory variable set to some value, say zero, just before we enter the inner loop. Any time we exchange the contents of two bins, we could set that variable to some other value, say one. Then, after we exit the inner loop, we could check to see if the memory variable was changed. If it was, we go back and repeat the outer loop. If it wasn't, we are through.



RDA126-48

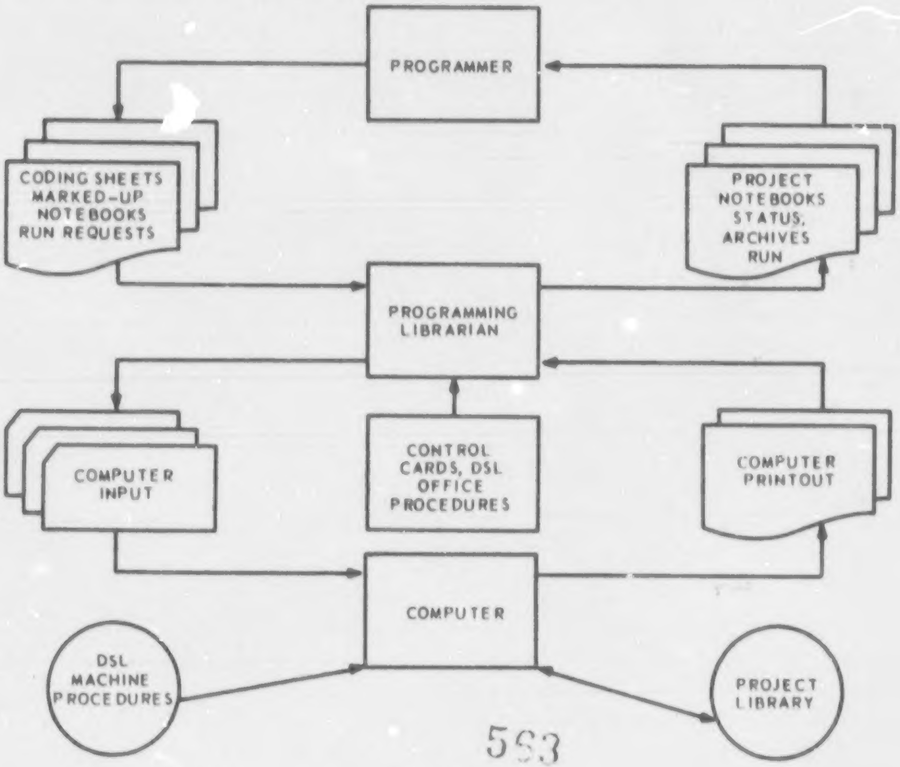
DEVELOPMENT SUPPORT LIBRARY

The Development Support Library (DSL) serves as a central repository of all data relevant to the project, in both human readable and machine recognizable form. As such, it is used to organize and control the software development and is the focal point of information exchange - both management and technical - for the life of the project.

The principal objective of the library is to provide constantly up-to-date representations of the programs and test data in both computer and human readable forms. The DSL concept is designed to separate the clerical and developmental tasks of programming. In addition, the DSL makes the code produced more visible to the team members.

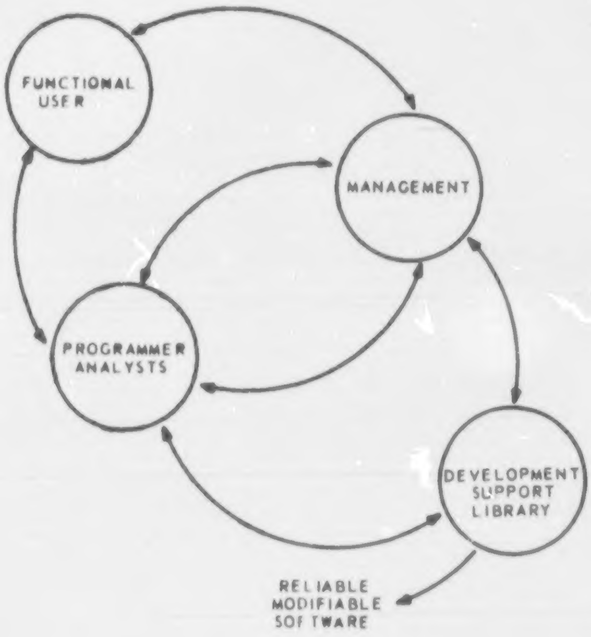
The components of a DSL, as an information base, are comprised of the internal and external libraries. The internal library consists of machine readable source programs, relocatable modules, object modules, linkage-editing statements, test data or job control statements. The external library consists of all current listings of programs, as well as listings of recent versions of the programs.

In many projects a development support library is maintained by a librarian who interfaces directly with the computer. Programmers interface directly with the computer only on an exception basis. In order to permit this, a standard set of procedures (the computer or machine procedures) for performing all machine operations is required. These procedures contain all the necessary information for updating libraries, link-editing jobs, and test runs, compiling modules and storing the object code and backing up the libraries. By using these procedures, the librarian is able to perform any of the library operations without direct assistance. Other team members communicate with the librarian in such ways as submitting original coding sheets, making notations on directories, and indicating changes on source listings.



The DSL provides a significant aid for testing and evaluation in that the code is centralized to avoid ambiguity of what is, and what is not, valid software. A development support library will normally consist of a production library which contains code that has been tested and one or more development libraries for new code. At any point of the project, the overall production library constitutes the current operational system. Therefore considerable care is taken to see that new segments and data item definitions have been properly tested before they are added. This testing is performed in the development libraries where segments are created as needed and exist until the units have been tested and added to the production library. When a segment is added to the production library, it is removed from the development library. More leniency is allowed in adding to a development library than in adding to the production library. For example, if a segment references a data item for which it is not authorized, it cannot be added to the production library. Unauthorized access is permitted in a development library, although the user would be warned that he has committed an apparent error. Control is obtained by requiring that an update to the production library be conditioned on proof of successful testing in a development library. This will reduce the likelihood of errors getting into the system. The verification procedures are reviewed by the manager whose approval should be required for update to the production library.

The development support library provides the necessary control for programming of a system in a top-down manner. Testing and integration will start with the highest level system segment as soon as it is coded. Since this segment will normally invoke or include lower level segments, code must exist for the next lower level segment. This code, called a program stub, may immediately return control, may output a message for debugging purposes each time it is executed, or may provide a minimal subset of the functions required. These program stubs are later expanded into full functional



segments, which in turn require lower level segments. Integration is, therefore, a continuous activity throughout the development process. During testing, the system executes the segments from the library that have been completed and uses the stubs where they have not. It is this characteristic of continuous integration that reduces the need for special test data drivers. The developing system itself can support testing because the code that interfaces with the newly added segments has previously been integrated and tested and can be used to feed test data to the new segments.

Program stubs can often be created as an automatic function of the development support library. This automatic function is provided by the programming support library (PSL). The PSL is a software system which provides the tools to organize, implement, and control computer program development. The system is designed specifically to support top-down development and structured programming. Different implementations of a PSL exist for various computer and operating system environments used in system development. The fundamental correspondence between the internal and external libraries in each environment is established by the PSL office and computer procedures. The office procedures are specified at a detailed level so that the format of the external libraries will be standard across programming projects, and the maintenance of both internal and external libraries can be accomplished as clerical functions. The PSL computer procedures for each are expressly designed for easy invocation by librarian personnel so that their use is nearly fail-safe.

The use of the top-down approach with a library provides a basis for capturing performance data during the development cycle. By replacing each stub with a timing loop that utilizes the estimated run time for that function, the developing system becomes a model. As dummy routines are replaced with working code, the performance results can be appraised against the performance objectives. In a similar manner, storage allocation can be modeled.

The use of a development support library combined with structured programming, top-down development, and HIPO diagrams significantly improves management control of the software development effort by providing continuous product visibility. Since the developing system is undergoing continuous integration, the system status is accurately reflected through the contents of the library; i.e., completeness is measured objectively in terms of how much of the system is operational. The completed code can be reviewed to verify status and appraise the quality of the software product.

TEAM OPERATIONS

Team operations represents a change in approach from a loosely structured group of programmers to a highly structured team of programming specialists who work under strict operational discipline. Teams organized in this fashion have demonstrated that they can produce quality code in a very efficient manner. The techniques of structured programming, top-down development, and development support libraries are always used. A team consists of the following positions:

1. Chief Programmer - A senior level programmer and analyst who is responsible for the development of the programming system in all respects. This person carries technical responsibility for the project including higher echelon coordination. He produces the critical core of the programming system in detailed code himself, directly specifies all other code required for system implementation, and reviews and oversees the integration of that code.

2. Backup Programmer - A senior level programmer and analyst who functions in full spectrum support of the chief programmer at a detailed task level so that he is constantly in position to assume the chief programmer's responsibility temporarily or permanently. He may be called upon to explore alternative design approaches, independent test planning, or other special tasks, but serves normally as an active participant in technical design, internal supervision, and external management functions.

3. Librarian - A programmer technician or secretary who has received additional training. Although he assembles, compiles, linkage-edits, and test-runs programs as requested by project programmers, the librarian is not simply a pooled assistant but a full-fledged member, with direct responsibility for the project-critical task of maintaining the library.

4. Team Members - Each team is a flexible module that can be supplemented with additional programmers, analysts, or technicians commensurate with the workscope. Depending on the size and nature of the programming project, either additional programmers can be added to a given team to write the programs specified by the chief programmer, or components of the overall design can flow to other teams for more detailed design and coding.

The term "senior level programmer" is, of course, relative; but within any programming production environment or agency, it can be applied to those assigned professionals whose technical competence, not only in programming details and techniques but also in broad system analysis and design, has distinguished them as programming problem-solvers. Technically, the chief programmer and the backup programmer should have such intimate familiarity with the resources and tools of the programming system and language with which they are working as to exploit them for specific program design purposes at the detailed coding level. On the management side, they must be capable of interfacing with the system project manager and translating his directives into programming production designs and plans that meet budget, time, and capability requirements. On some teams, the chief programmer is the first-line manager, while on other teams he is the technical leader who works in close liaison with the first-line manager.

Reintroducing senior people such as the chief and backup programmers into detailed program coding recognizes a new set of circumstances in today's operating system environment. The job control language, data management access methods, utility facilities, and high-level source languages are so rich that there is both a need and an opportunity for using senior personnel at the detailed coding level. The need is to make the best possible use of a very extensive set of facilities. The functions of the onerating systems are impressive, but they are called into play by language forms that require a good deal of study and experience to utilize in the most effective manner. Likewise, the opportunity exists for a good deal of work reduction and simplification in the application being written, in both original programming and later maintenance. For example, the intelligent use of a high-level data management capability may eliminate the need to develop a private file processing system. Finding such an intelligent use is not an easy task, but it can bring substantial reduction in code required and easier system maintenance.

The job structuring in team operations isolates functional responsibilities between data definition, program design, clerical operations, etc., so that accountabilities are better defined. Consequently, communication among team members is sharpened and more precise. The very separation of skills forces a high degree of public practice. For example, the librarian is responsible for picking up all computer output, good or bad, and filing it in the notebooks and archives of the development support library where it becomes part of the public record. By contrast, in traditional programming operations the bad runs go into the wastebasket, often destroying information of latent value, but certainly destroying information about errors of carelessness. This identification of all program data and computer runs as public assets, not private property, is a key principle in team operations.

Team operations allows for professional growth and technical excellence in programming. Since delegated clerical procedures are used to maintain programming system development in a highly structured and visible form, more time and energy can be allocated to developing key technical skills and building the desirable software system. This creative environment provides good training for other programmers associated with a team, preparing them for leadership in future teams. Inefficient coding habits and techniques can easily be identified and corrected.

SUMMARY

The techniques described in this document represent a disciplined approach to application development which can have a dramatic impact on the data processing department's ability to respond to its users. The use of this improved technology should result in improvements in manageability, productivity, and program quality and maintainability.

The use of structured programming brings a higher precision and reliability to programming than ever before. It results in programs which can be read, maintained, and modified by other programmers. Top down development imposes an architectural discipline which reduces the traumas of integration testing, and promotes a more orderly system development. HIPO supports top down development and provides meaningful documentation that keys on function. The job structuring within team operations isolates functional responsibilities between data definition, clerical operations, program design, etc., so that accountabilities are better defined. As a result, all team members are motivated to think and communicate more accurately and consistently about their specific jobs. Structured walk-throughs locate design and coding errors much earlier in the process and enable management to convert project reviews into meaningful milestones which contribute in themselves to team productivity. Finally, a development support library provides the visibility needed to measure and control the development process and helps the data processing department, like its users, take advantage of the computer's time and labor saving powers.

REFERENCES

Baker, F. T. "Chief Programmer Team Management of Production Programming." IBM Systems Journal, Volume 11, No. 1, 1972.

Baker, F. T. "System Quality Through Structured Programming." In AFIPS Conference Proceedings, Volume 41, Part I, 1972.

Bohm, C., and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." Communications of the ACM 9, No. 3 (May 1966), 366-371.

Dijkstra, E. W. "Notes on Structured Programming." THE (Technische Hogeschool Eindhoven), Netherlands, Second Edition, 1970.

Mills, H. D. Chief Programmer Teams: Principles and Procedures. International Business Machines Corporation, FSC 71-5108, June 1971.

Mills, H. D. Mathematical Foundations for Structured Programming. International Business Machines Corporation, FSC 72-6012, February 1972.

Mills, H. D. "Top-Down Programming in Large Systems." Courant Computer Sciences Symposium 1, New York University, June 1971, edited by Randall Rustin, Prentice-Hall (1971), 41-55.

Weinberg, G. M. The Psychology of Computer Programming. New York: Van Nostrand Reinhold, 1971.

STRUCTURED WALK-THROUGHS

Project management has long recognized the need for periodic reviews as a vehicle for determining where the project stands in relation to its schedule, and for identifying areas that require special attention. Generally, however, these exercises have been looked upon with misgivings by those who must submit themselves to the review.

The situation which classically arises during the review is one of conflict and hostility. The review takes on the appearance of a witch hunt, and the reviewer finds himself in the position of inquisitor. At best, the reviewees feel they have little to gain from this encounter and most probably feel that they will come out of the review with a list of "to-dos" which will only serve to put them farther behind in their development schedules. More damaging still is their belief that the longer the list, the longer the indictment against them. They feel that they will learn nothing in the review which will help them attack their unique problems; moreover, they feel that they will spend a large and unproductive portion of the meeting just bringing the reviewer up from ground zero.

The structured walk-through described here increases the value of these reviews beyond a determination of schedule variance and problem identification, and eliminates many of the negative aspects. Within IBM the structured walk-through is:

1. A positive motivator for the project team.
2. A learning experience for the team.
3. A tool for analyzing the functional design of a system.
4. A tool for uncovering logic errors in program design.
5. A tool for eliminating coding errors before they enter the system.
6. A framework for implementing a testing strategy in parallel with development.
7. A measure of completeness.

A structured walk-through is a generic name given to a series of reviews, each with different objectives and each occurring at different times in the application development cycle. The basic characteristics of the walk-through are:

1. It is arranged and scheduled by the developer (reviewee) of the work product being reviewed.

2. Management does not attend the walk-through and it is not used as a basis for employee evaluation.

3. The participants (reviewers) are given the review materials prior to the walk-through and are expected to be familiar with them.

4. The walk-through is structured in the sense that all attendees know what is to be accomplished and what role they are to play.

5. The emphasis is on error detection rather than error correction.

6. All technical members of the project team, from most senior to most junior, have their work product reviewed.

Mechanics

The objectives of the structured walk-through will be different at different stages of the project. The basic mechanics will, however, remain the same. The reviewee, the person whose work product is being reviewed, is responsible for arranging the meeting. Several days prior to the meeting, the reviewee selects the attendees he feels are required, distributes his work product to them, states what the objectives of the walk-through will be, and specifies what roles the reviewers are to play.

Although there are no hard and fast rules as to who the reviewers should be, the idea is for the reviewee to pick those interested parties who can detect deviations, inconsistencies, and violations within the work product or in the way that it interacts with its environment. Typically, but not necessarily, the reviewers will be project teammates of the reviewee. For example, early in the project, when a major objective is to insure that the system is functionally complete, the reviewee might want user representatives. Or, if programmers and analysts are functionally separated, and the objective of the walk-through is to insure that the programmer's internal specifications match the analyst's external specifications, then the programmer would want the analyst to attend. Within IBM, it is not uncommon for a programmer to reschedule a walk-through several times in order to insure that a particular reviewer will be available.

A typical walk-through will include four to six people and will last for a prespecified time, usually one or two hours. If at the end of that time the objectives have not been met, another walk-through is scheduled for the next convenient time. Someone is designated as the recording secretary. This person records all the errors, discrepancies, exposures, and inconsistencies that are uncovered during the walk-through. This record becomes an action list for the reviewee and a communication vehicle with the reviewers.

In addition to the substantive questions which will hopefully arise in the reviewer's mind prior to the walk-through, he will undoubtedly detect minor mistakes such as typographical, spelling, grammatical, and coding syntax errors. These can be handled several ways. One way is to instruct each reviewer to make an error list and pass it to the recording secretary at the beginning of the walk-through. Another way is for each reviewer to cover these errors with the reviewee offline. Or, the reviewers can annotate their copies of the work product and return it to the reviewer at the end of the walk-through. The important point is that the walk-through should be concerned with problems of greater substance (i.e., ambiguous specifications, basic design flaws, poor logic, inappropriate or inefficient coding techniques).

Mechanically, what takes place during the structured walk-through? First, the reviewers are requested to comment on the completeness, accuracy, and general quality of the work product. Major concerns are expressed and identified as areas for potential follow-up. The reviewee then gives a brief tutorial overview of the work product. He next "walks" the reviewers through the work product in a step-by-step fashion which simulates the function under investigation. He attempts to take the reviewers through the material in enough detail so that the major concerns which were expressed earlier in the meeting will either be explained away or brought into focus. New thoughts and concerns will arise during this "manual execution" of the function, and the ensuing discussion of these points will crystallize everyone's thinking. Significant factors that require action are recorded as they emerge.

A key element regarding the structured walk-through is its relationship to the project test strategy. Within IBM, the structured walk-through is part and parcel of a parallel test strategy and, in fact, the "manual execution" is often driven by formalized test cases.

Immediately after the meeting, the recording secretary distributes copies of the handwritten action list to all the attendees. It is the responsibility of the reviewee to insure that the points of concern on the action list are successfully resolved, and that the reviewers are notified of the actions taken and/or the corrections that have been made. (This latter point is important because many of the revelations which arise impact the reviewers, particularly if they and the reviewee are teammates.) Management does not double-check the action list to insure that the outstanding problems have been resolved, nor does it use this list as a basis for employee evaluation. Rather, the action list is considered to be a tool used to improve the product.

As Part of New Technologies

Structured walk-throughs have been implemented within IBM programming groups which are using structured programming, top down development, development support libraries, and team operations. In fact, the use of walk-throughs as described in this text has evolved to its present position because of these new technologies.

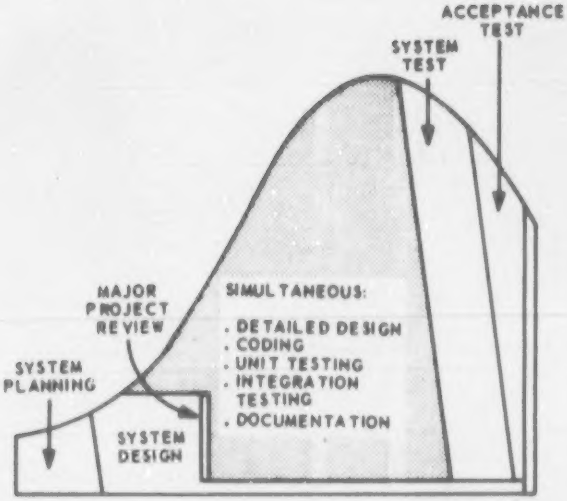
The visibility inherent in structured programming, the idea that code is meant to be read by others, the enforced programming conventions, and the simplified program logic make it easy for the reviewer to be "walked through" code segments.

The use of HIPO as a top down design and documentation tool lends itself well to the structured walk-through. HIPO's graphical representation of function gives the reviewee the luxury of something concrete and tangible through which he can take the reviewers in a step-by-step fashion at increasing levels of detail.

A development support library organizes and structures the emerging system so that the details can be easily reviewed. In addition, the librarian can also serve as the recording secretary for the walk-throughs.

The concept of a tightly knit team, whose members possess unique skills and who are in close communication with each other, is logically supported by the idea of a structured walk-through. Since the chief programmer and the backup programmer already read code, the extension to everyone reading code is not a major jump. Additionally, there is value in the walk-through as an educational tool. Because the chief programmer and the backup programmer design and code the top of the system first, their initial walk-throughs serve as important learning experiences for the other team members--both in terms of design and coding techniques and as an introduction to the system.

Within an application development cycle, there are several major milestones and many minor milestones where the walk-through technique can be used. As an example, a manning curve for an application development cycle in which the new technologies are being used might look as shown in Figure 4. The management of this project could decide that one condition for successfully reaching the milestones, listed in the left-hand column of Figure 5 is that the items in the right-hand column must have been reviewed in a structured walk-through. In this sense, the walk-through tracks progress and serves as a meaningful measure of completeness.



RDA126-51

Figure 4. A Typical Manning Curve for an Application Development Cycle. Major Milestones Where Structured Walk-Throughs Might Be Employed Include End of System Planning, End of System Design and End of Development.

		PROJECT MILESTONES	ITEMS TO BE REVIEWED VIA A STRUCTURED WALK-THROUGH
Major Project Milestones		End of System Planning	Project Plans System Definition Task Identification
	Multiple Minor Milestones	Major Project Review "Technical"	Functional Specifications Work Assignments Schedules
		Detailed Design	Internal Specifications HIPO Package
		Coding	Uncompiled Source Listings
		Documentation	User Guides Programmer Maintenance Manuals . Internal Specifications . HIPO Package
		End of Development	Deliverable Product . Code . Documentation

RDA126-52

Figure 5. The Table Shows Items Which Might Be Reviewed Using Structured Walk-Throughs at Various Times During a Project. The Minor Milestones Would Be Repeated as the System Grew.

Parallel Testing

The structured walk-through can serve to establish a framework for parallel testing. Parallel testing implies: (1) the development of test cases and testing procedures in parallel with the development of the system, and (2) an independent tester who is responsible for implementing the test strategy.

When using team operations, the tester would logically be the backup programmer. In large, functionally separated organizations, the tester(s) might come from an independent group.

The tester builds a product in much the same manner as the developer does. They both start at the same place with a set of functional specifications. The developer, however, looks at the specs as a builder might look at blueprints, while the tester looks at those specs in the way a building inspector might look at blueprints. The tester, like the inspector, attempts to insure that the specifications meet certain standards, and that the product matches the specifications.

A functional program specification can be boiled down to a set of cause and effect relationships:

- "If the accumulated FICA deduction is equal to or greater than $10,800, then return the difference to net pay."
- "When the on-hand balance falls below the reorder point, transfer control to the EOQ routine."
- "Set the transmission line to inoperative and notify the network control operator if the retry procedure fails."

Initially, the tester takes the functional specifications and breaks them down into a series of cause and effect statements. Rigorous testing means that each of these cause and effect relationships must be tested. That is to say, the tester, using some form of tabular or graphical assistance, must determine whether each cause has its desired effect. Unfortunately, this is not always easy to do. If it were, testing would not be a problem and systems would be more error-free. Cause and effect relationships tend to string together in complex logical chains. Therefore, it is not always obvious what is a cause and what is an effect. In addition, analysts and designers do not apply the same discipline to their specifications that the programmer must apply to his code. Rather, they tend toward free-flowing prose, resplendent with inconsistencies.* Nevertheless, the product which the tester is creating will evolve into a formalized set of machine-readable test cases, residing in a test library which, based on the quality of his efforts and the thoroughness with which he breaks down the functional specifications, will test the code.

Within IBM, the tester plays a key role in those structured walk-throughs which relate to detailed design and programming. The tester views the walk-through as the vehicle which formally brings him together with the developer. After the reviewee walks the reviewers through the work product to bring everyone to a common level of understanding, he passes control of the meeting to the tester. The tester presents his test cases, one by one, to the reviewee. All participants observe as the reviewee walks each test case through the work product. Inconsistencies and errors are spotted in the work product and also in the test cases. The recording secretary is responsible for recording problems that relate to the product, and the tester is responsible for recording and correcting problems that relate to his test cases. The tester's goal is to produce a complete and non-overlapping library of test cases which will validate the final product.

---

* The English language is not noted for its ability to express complex relationships with precision. Perhaps the future will see us evolve into structured specification languages. A step in that direction would be pseudo code narrative associated with structured programming.

The evolution of the test library proceeds in parallel with the system. While the system develops from functional specifications to internal program specifications and HIPO diagrams, to source code, and finally to compiled code, the tester is independently developing the test library from the functional specifications, to cause and effect relationships, to manual test cases, and finally to machine-readable test cases. By the time a subset of the system is ready to be compiled, the test cases will be included in the test library and can be driven against the compiled code.

This parallel evolution of the application and its test cases, synchronized at each development step by a structured walk-through, insures a thoroughness and a discipline which cannot be achieved when testing is handled as a follow-on to development.

Psychology

The interested reader may wonder why management doesn't take a more active part in the walk-through or, more specifically, why management doesn't use the action list as a measure of employee performance. The answer is that management could, but only at the expense of losing some of the values of the walk-through.

An essential ingredient for a successful walk-through is an open and non-defensive attitude on the part of the participants. A productive atmosphere is one in which the reviewee makes it easy for the reviewers to find problems. He should welcome their feedback and should encourage their frankness. If, however, he feels that he is being evaluated by what occurs in the walk-through, and by the size of the action list, he will naturally tend to suppress criticisms. He will be defensive and unreceptive to new ideas. His ego will be staked to the work product, and he will have little motivation to use the session as a learning experience. A successful walk-through, by comparison, is one in which many errors and inconsistencies are uncovered.

The role of the reviewers is one of preparation, non-malicious probing, and problem definition. If they are teammates of the reviewee, it will not be uncommon for them to discover that hidden relationships exist between what they are developing and what is being reviewed. Ambiguities will come to light which will require further clarification and definition. If for no other reason, management should value the walk-through for its contribution as a communication tool among the developers.

Setting the proper psychological atmosphere for structured walk-through is the key. An organization utilizing team operations, top-down development, and structured programming can do it rather naturally. Since the chief programmer and the backup programmer will produce the initial design and the most critical code in the system, their work products will be the first under review. Because they are more senior and more closely attuned to management's desires (the chief programmer may in fact be the manager), they are in a position to establish the proper framework and attitude surrounding the walk-through. In addition, these initial walk-throughs will serve as a learning experience for the team not only as to the walk-through mechanics, but with respect to the system itself.

SUMMARY

Our experience with structured walk-throughs has been most encouraging. Undoubtedly, there are a number of ways they could be modified to fit other organizations. The central idea, however, should remain the same; i.e., to convert the classical project review into a productive working session which not only tracks progress, but which makes a positive contribution to that progress. Outwardly, management involvement appears low, but in reality structured walk-throughs provide management with a vehicle for catching errors in the system at the earliest possible time when the cost of correcting them is lowest and their impact is smallest.

THE IDEA OF STRUCTURED PROGRAMMING

It has been discovered recently that computer programs can be written with a high degree of structure, which permits them to be more easily understood for testing, maintenance, and modification. With structured programming, control branching is entirely standardized so that code can be read from top to bottom, without having to trace the branching logic as is typical for code generated in the past. Structured programming represents a new technical standard which permits better enforcement of design quality for programs. It corresponds to principles in hardware design where it is known that all possible logic circuits can be formed out of a small collection--AND, OR, NOT--of standard component circuits.

In structured programming, programmers must think deeper, but the end result is easier to read, understand, and maintain. The standards of structured programming are based on new mathematical theorems, and do not require case by case justification. Just as it is the burden of a professional engineer to be able to design logic circuits out of certain basic components, so it is the burden of a professional programmer to write programs in a structured way, using only recently standardized branching conventions.

Top Down Programming

Structured programming also enhances the development of programs in a "top down" form, in which major programs can be broken into smaller programs through a combination of code and the designation of dummy programs called program stubs which are referenced or called by that code. By writing the code which calls the program stubs before the stubs themselves are developed, the interfaces between the calling and the called programs are defined completely so that no interface problems will be encountered later.

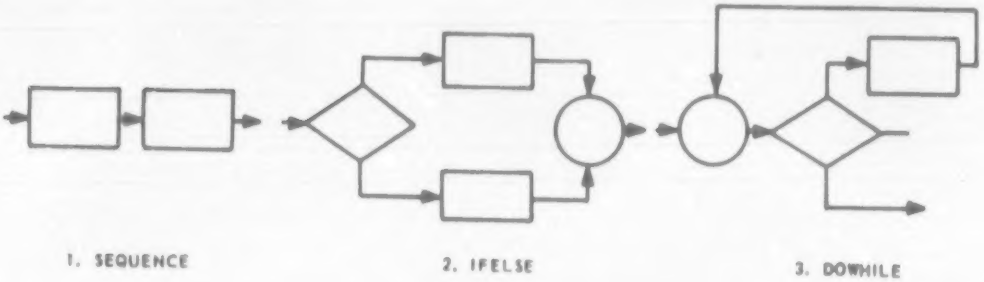
The result of the systematic disciplined approach of structured programming is higher precision programming than was possible before. The testing of such programs is accomplished more rapidly, and the final results are programs which can be read, maintained, and modified by other programmers with much greater facility.

Structured Programming Theory

Any program, no matter how large or complex, can be represented as a set of flowcharts. Structured programming theory deals with converting large and complex flowcharts into standard forms so that they can be represented by iterating and nesting a small number of basic and standard control logic structures.

A sufficient set of basic control logic structures consists of three members (see Figure 6):

1. A sequence of two operations (MOVE, ADD, . . .).
2. A conditional branch to one of two operations and return (an IFELSE statement).
3. Repeating an operation while some condition is true (a DOWHILE statement).



RDA126-53

Figure 6. Basic Control Logic Structures

The basic structure theorem, due in original form to Bohm and Jacopini*, is that any flowchart can be represented in an equivalent form as an iterated and nested structure in these three basic and standard figures.

Note that each structure has only one input and one output, and can be substituted for any box in a structure, so that complex flowcharts can result. The key point is that an arbitrary flowchart has an equivalent representative in the class so built up.

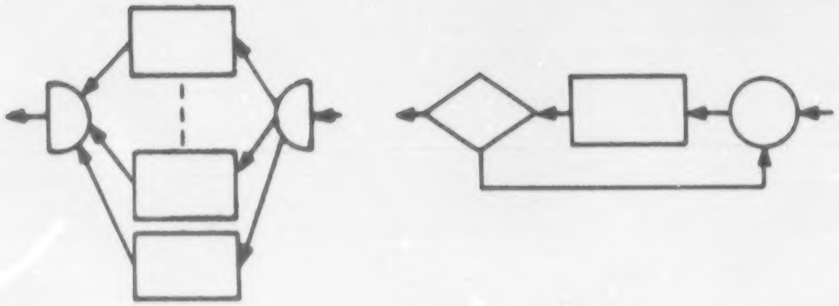
The structure theorem demonstrates that programs can be written in terms of IFELSE and DOWHILE statements. The idea of an unconditional branch and corresponding statement label is never introduced in these basic structures, and is thus never required in a representation.

There is no compelling reason in programming to use such a minimal set of basic figures, and it appears practical to augment the basic set with two variations in order to provide more flexibility. The variations, DOUNTIL and CASE, are shown in Figure 7.

DOUNTIL provides an alternative form of looping structure, while CASE is a multi-branch, multi-join control structure in which it is convenient to express the processing of one of many possible unique occurrences.

---

* Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of the Association for Computing Machinery, Volume 9, No. 4, May 1966.



4. DOUNTIL

5. CASE

RDA126-54

Figure 7. Variations of Basic Control Logic Structures

A major characteristic of programs written in these structures is that they may be literally read from top to bottom; there is never any "jumping around" as is so typical in trying to read code which contains unconditional branches. This property of readability is a major advantage in developing, testing, maintaining, or otherwise referencing code at later times.

Another advantage, of possibly even greater benefit, is the additional program design work that is required to produce such structured code. The programmer must think through the processing problem, not only writing down everything that needs to be done, but writing it down in such a way that there are no afterthoughts with subsequent jump-outs and jump-backs and no indiscriminate use of a section of code from several locations because it "just happens" to do something at the time of the coding. Instead, the programmer must think through the control logic of the module completely at one time to provide the proper structural framework for the control. This means that programs will be written in a much more uniform way because there is less freedom for arbitrary variety.

Such a program is much easier to understand than an unstructured logical jumble; readability has been improved. Because of its simplicity and clear logic, it minimizes the danger of the programmer's overlooking logical errors during implementation; reliability has been improved. Improved reliability, in combination with the great simplicity obtained by structured programming, naturally leads to improved maintainability. Further, because structured code is simple, a programmer can control and understand a much larger amount of code. With increased productivity, programming costs can be reduced.

Segmenting Structured Programs

Imagine a 100-page program written in structured code. Although it is highly structured, such a program is still not very readable. The extent of a major DO loop may be 50 or 60 pages, or an IFELSE statement may take 10 or 15 pages. This is simply more than the eye can comfortably take in or the mind retain for the purpose of programming.

However, with our program in structured form, we can begin a process which we can repeat over and over until we get the whole program defined. This process is to formulate a 1-page skeleton program which represents that 100-page program.

We do this by selecting some of the most important lines of code in the original program and then filling in what lies between those lines by names. Each new name will refer to a new segment to be stored in a library and called by a macro facility insert. In this way, we produce a program segment with something under 50 lines, so that it will fit on one page. This program segment will be a mixture of control statements and macro calls with possibly a few initializing, file, or assignment statements as well.

The programmer must use a sense of proportion and importance in identifying what is the forest and what are the trees out of this 100-page program. It corresponds to writing the "high level flowchart" for the whole program, except that a completely rigorous program segment is written. A key aspect of structured programming is that in any segment referred to by name, control enters at the top and exits at the bottom, and has no other means of entry or exit from other parts of the program. Thus, when reading a segment name, at any point, the reader can be assured that control will pass through that segment and not otherwise affect the control logic on the page he is reading.

To satisfy the segment entry/exit requirements, we need only be sure they include all matching control logic statements on a page. For example, the ENDO to any DO and the ELSE to any IF should be put in the same segment.

For the sake of illustration, this first segment may consist of some 30 control logic statements, such as DOWHILEs, IFELSEs, perhaps another 10 key initializing statements, and some 10 macro calls. These 10 macro calls may involve something like 10 pages of programming each for the original 100 pages, although there may be considerable variety among their sizes.

Now we can repeat this process for each of these 10 segments. Our end result is a program which has been organized into a set of named member segments, each of which can be read from top to bottom without any side effects in control logic other than what is on that particular page. A programmer can access any level of information about the program, from highly summarized at the upper level segments, to complete details in the lower levels.

In the preceding paragraphs, we assumed that a large structured program somehow existed, already written with structured control logic, and discussed how we could conceptually reorganize the identical program in a set of more readable segments. In the following text, we observe how we can create such structured programs a segment at a time in a natural way.

Creating a Structured Program

Suppose that a program has been well designed, and that we are ready to begin coding. Also note that a common pitfall in programming is to "lose our cool" --i.e., begin coding before the design problems have been thought through well enough. In this case, it is easy to compromise a design because code already exists which isn't quite right, but "seems to be running correctly." The result is that the program gets warped around code produced on the spur of the moment.

Our main point is to observe that the process of coding can take place in practically the same order as the process of extracting code from our imaginary large program in the previous section. That is, armed with a program design, one can write the first segment which serves as a skeleton for the whole program, using segment names, where appropriate, to refer to code that will be written later. In fact, by simply taking the precaution of inserting dummy members into a library with those segment names, one can compile or assemble, and even possibly execute, this skeleton program while the remaining coding is continued. Very often, it makes sense to put a temporary statement "Got to here OK" as a single executable statement in such a dummy member.

Now, the segments at the next level can be written in the same way, referring as appropriate to segments to be later written and setting up dummy segments as they are named in the library. As each dummy segment becomes filled in with its code in the library, the recompilation of the segment that includes it will automatically produce updated, expanded versions of the developing program. Problems of syntax and control logic will usually be isolated within the new segments so that debugging and checkout goes correspondingly well with such problems so isolated.

It is clear that the programmer's creativity and sense of proportion play a large factor in the efficiency of this programming process. The code that goes into earlier sections should be dictated, to some extent, not only by general matters of importance, but also questions of getting executable segments reasonably early in the coding process. For example, if the control logic of a skeleton module depends on certain control variables, their declarations and manipulations may want to be created at fairly high levels in the hierarchy. In this way, the control logic of the skeleton can be executed and debugged, even in the still skeleton program.

Note that several programmers may be engaged in the foregoing activity concurrently. Once the initial skeleton program is written, each programmer could take on a separate segment and work somewhat independently within the structure of an overall program design. The hierarchical structure of the programs contribute to a clean interface between programmers. At any point in the programming, the segments already in existence give a concise framework for fitting in the rest of the work.

APPENDIX A

(Extract from Chapter 6 of AFM 171-10, Vol I)

Chapter 6

FLOWCHART SYMBOLS FOR DATA PROCESSING

020601. General. The purpose of this chapter is to establish flowchart symbols for use in the preparation of flowcharts for automatic data processing systems and applications. These symbols are the American Standard Flowchart Symbols which were approved by the Department of Defense.

020602. Responsibility. It is mandatory that the American Standards Association symbols be used by the Air Force in the preparation of all new and revised ADPS flowcharts. Existing flowcharts need not be reaccomplished for the sole purpose of converting to the American Standards Association symbols.

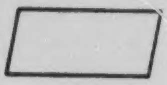
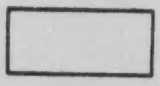
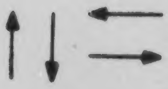
020603. Flowchart Symbols.

a. Symbols Represent Functions. Symbols are used on a flowchart to represent the functions of a data processing system. These functions are INPUT/OUTPUT, PROCESSING, FLOW DIRECTION, and ANNOTATION.

A basic symbol is established for each function and can always be used to represent that function. Specialized symbols are established which may be used in place of a basic symbol to give additional information.

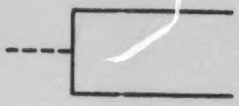
The size and the dimensional ratio of each symbol may vary depending on its specific use but not to the point of losing its identity.

b. Basic Symbols.

Symbols	Descriptions
	Input/Output Symbol. The symbol shown represents the input/output function (I/O); i.e., the making available of information for processing (input) or the recording of processed information (output).
	Processing Symbol. The symbol shown represents the processing function; i.e., the process of executing a defined operation or group of operations resulting in a change in value, form, or location of information, or in the determination of which of several flow directions are to be followed.
	Flow Direction Symbol. The symbols shown represent the flow direction function; i.e., the indication of the sequence of available information and executable operations. Flow direction is represented by lines drawn between symbols. Normal direction flow is from top to bottom and left to right. When the flow direction is not top to bottom and left to right, open arrowheads shall be placed on reverse

Symbols **Descriptions**

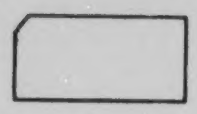
direction flowlines. When increased clarity is desired, open arrowheads can be placed on normal direction flowlines. When flowlines are broken due to page limitation, connector symbols shall be used to indicate the break. When flow is bidirectional, it can be shown by either single or double lines but open arrowheads shall be used to indicate both normal direction flow and reverse direction flow.



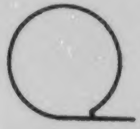
Annotation Symbol. The symbol shown represents the annotation function; i.e., the addition of descriptive comments or explanatory notes as clarification. The broken line may be drawn either on the left as shown or on the right. It is connected to the flowline at a point where the annotation is meaningful by extending the broken line in whatever fashion is appropriate.

c. Specialized Input/Output Symbols. Specialized I/O symbols may represent the I/O function and, in addition, denote the medium on which the information is recorded or the manner of handling the information or both. If no specialized symbol exists, the basic I/O symbol is used. These specialized symbols are:

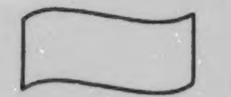
Symbols **Descriptions**



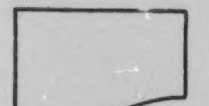
Punched Card Symbol. The symbol shown represents an I/O function in which the medium is punched cards, including mark sense cards, partial cards, stub cards, etc.



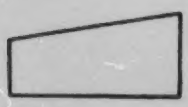
Magnetic Tape Symbol. The symbol shown represents an I/O function in which the medium is magnetic tape.



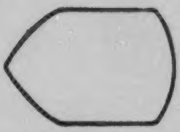
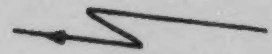
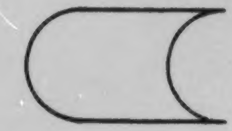
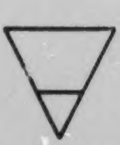
Punched Tape Symbol. The symbol shown represents an I/O function in which the medium is punched tape.



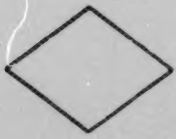
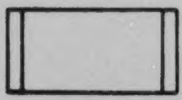
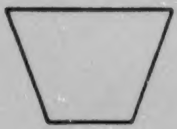
Document Symbol. The symbol shown represents an I/O function in which the medium is a document.

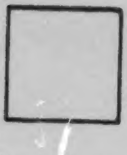
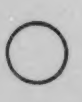
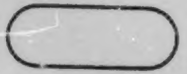


Manual Input Symbol. The symbol shown represents an I/O function in which the information is entered manually at the time for processing, by means of online keyboards, switch settings, pushbuttons, card readers, etc.

Symbols	Descriptions
	**Display Symbol.** The symbol shown represents an I/O function in which the information is displayed for human use at the time of processing, by means of online indicators, video devices, console printers, plotters, etc.
	**Communication Link Symbol.** The symbol shown represents an I/O function in which information is transmitted automatically from one location to another. To denote the direction of data flow, the symbol is always drawn with superimposed arrowheads.
	**Online Storage Symbol.** The symbol shown represents an I/O function utilizing auxiliary mass storage of information that can be accessed online; e.g., magnetic drums, magnetic disks, magnetic tape strips, automatic magnetic card systems or automatic microfilm chip or strip systems.
	**Offline Storage Symbol.** The symbol shown represents any offline storage of information, regardless of the medium on which the information is recorded.

d. **Specialized Processing Symbols.** Specialized processing symbols may represent the processing function and, in addition, identify the specific type of operation to be performed on the information. If no specialized symbol exists, the basic processing symbol is used. These specialized symbols are:

Symbols	Descriptions
	**Decision Symbol.** The symbol shown represents a decision or switching type operation that determines which of a number of alternate paths is to be followed.
	**Predefined Process Symbol.** The symbol shown represents a named process consisting of one or more operations or program steps that are specified elsewhere; e.g., subroutine or logical unit.
	**Manual Operation Symbol.** The symbol shown represents any offline process geared to the speed of a human being.

Symbols	Descriptions
	**Auxiliary Operation Symbol.** The symbol shown represents an offline operation performed on equipment not under direct control of the central processing unit.
	**Connector Symbol.** The symbol shown represents a junction in a line of flow. A set of two connectors is used to represent a continued flow direction when the flow is broken by any limitation of the flowchart. A set of two or more connectors is used to represent the function of several flowlines with one flowline or the junction of one flowline with one of several alternate flowlines.
	**Terminal Symbol.** The symbol shown represents a terminal point in a system or communication network at which data can enter or leave; e.g., start, stop, halt, delay, or interrupt.

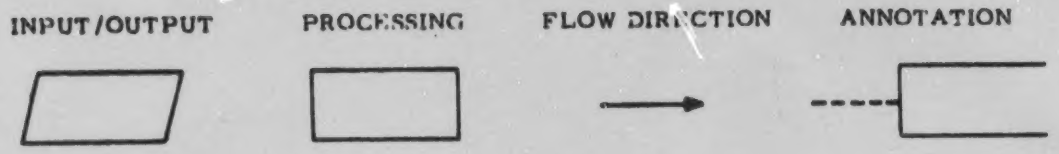
f. Existing flowchart templates, i.e., those provided by the manufacturers, may be utilized to form the flowchart symbols above.

020604. Summary of American Standard Flowchart Symbols.

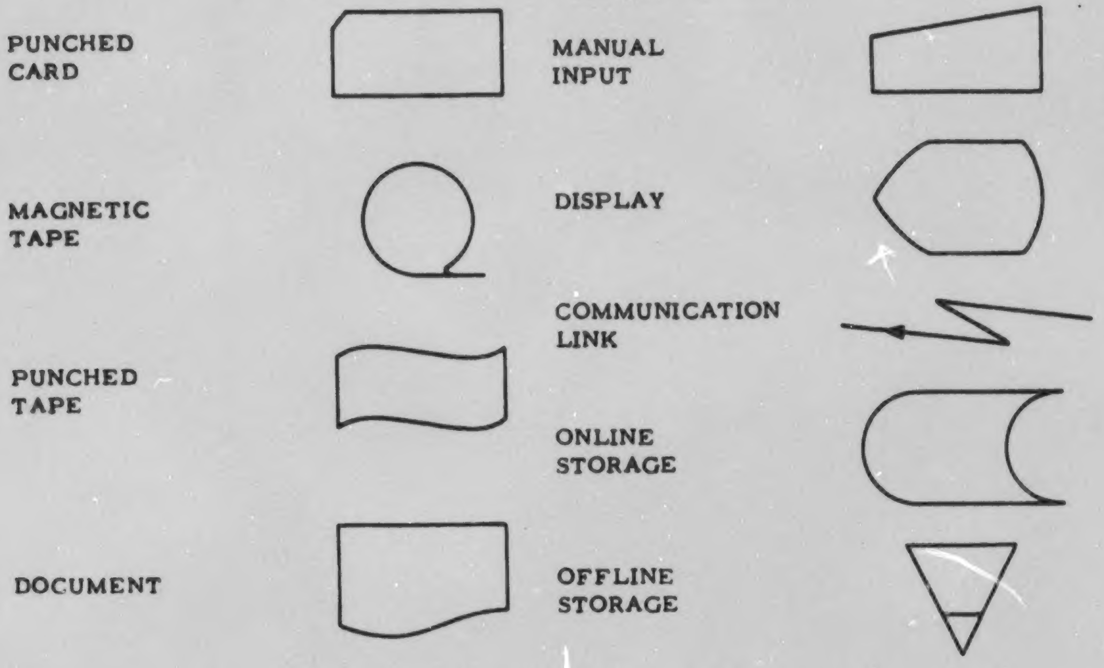
A summary of flowchart symbols is illustrated on the following page.

SUMMARY OF FLOWCHART SYMBOLS

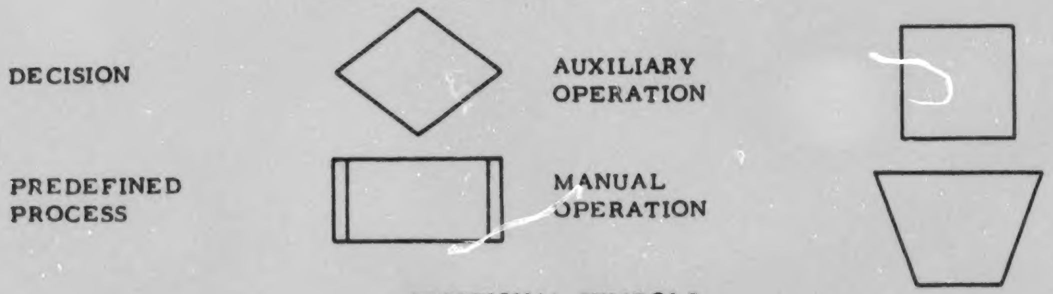
BASIC SYMBOLS



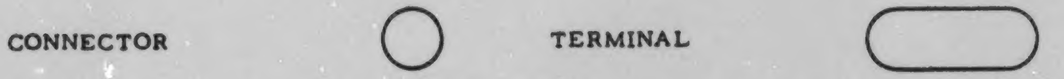
SPECIALIZED INPUT/OUTPUT SYMBOLS



SPECIALIZED PROCESSING SYMBOLS



ADDITIONAL SYMBOLS



STUDENT HANDOUT
3ABR51113TB
KDA-781

Technical Training

Programming Specialist (Honeywell)

FORTRAN LANGUAGE

January 1976

USAF SCHOOL OF APPLIED AEROSPACE SCIENCES
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

ATC [illegible]

DO NOT USE ON THE JOB

Table of Contents

Section 1 Introduction

Character Set 1-1

Programming Form

- Comments 1-1
- Continuations 1-2
- Labels 1-2
- END 1-2
- Blanks 1-2

Section 2 Constants, (etc.)

- Constants 2-1
- Variables 2-2
- Arrays 2-3
- Subscripts 2-4

Section 3 Specification Statements

- Dimension 3-2
- Common 3-4
- Equivalence 3-7
- External 3-9
- Type Statements 3-10
- Data 3-11

Section 4 Arithmetic Assignment Statements

- Operators 4-1
- Mixed Mode 4-1
- Parentheses 4-2

Section 5 Logical Assignment Statements

Relational Operators 5-1
Logical Operators 5-2
Evaluation of expressions 5-2

Section 6 Control Statements

Go To 6-2
Assigned Go To 6-3
Computed Go To 6-4
Arithmetic If 6-5
Logical If 6-6
Call 6-7
Return 6-8
Continue 6-9
Stop 6-10
Pause 6-11
Do 6-12

Section 7 Input-Output

non-Formatted Read 7-1
Formatted Read 7-2
non-Formatted Write 7-2
Formatted Write 7-2
Format Statement 7-3
Integer Descriptor 7-4
Real Descriptors 7-4
Double Precision Descriptors 7-6
Complex Conversion 7-7
Logical Descriptors 7-7

Blank Descriptor 7-7

Hollerith Descriptor 7-7

Alphanumeric Descriptor 7-8

Scaling 7-8

Repetition Constant 7-9

Scanning 7-10

Printer 7-10

Multiple Records 7-11

Short list 7-11

Format at Execution 7-12

Implied Do 7-12

Rewind 7-14

Backspace 7-14

Endfile 7-14

Section 8 Statement Functions

Naming 8-1

Dummy Arguments 8-1

Executing 8-2

Section 9 Subprograms

Block Data 9-2

Function Subprograms 9-3

Subroutine Subprograms 9-5

Section 10 Predefined Functions

Intrinsic Functions 10-1

Basic External Functions 10-1

Index

Introduction:

FORTRAN is a programming language similar to the language of mathematics. It is used primarily in mathematical and scientific applications.

This manual deals with standard FORTRAN IV as defined by the United States of America Standards Institute. There are many versions of FORTRAN, written expressly for particular computers. These FORTRAN's differ, more or less, from each other and from standard FORTRAN. This manual does not deal with the peculiarities of any FORTRAN system.

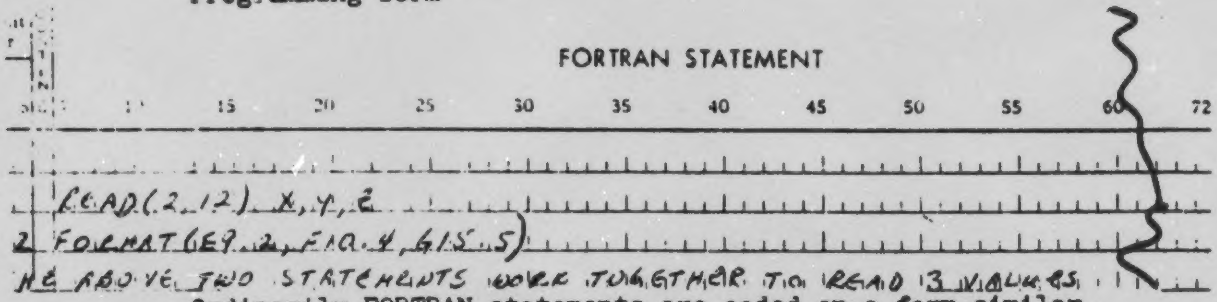
The term FORTRAN is also applied to the compiler program, or programs, which translates the FORTRAN source language into machine language.

The FORTRAN character set consists of 47 characters:

The alphabetics	A - Z
The numerics	0 - 9
blank	
equals	=
plus	+
minus	-
asterisk	*
slash	/
left parenthesis	(
right parenthesis	)
comma	,
decimal point	.
dollar sign	$

This order does not imply a collating sequence.

Programming form



```
  READ(2,12) X,Y,Z
2 FORMAT(E9.2,F10.4,I15.5)
HE ABOVE TWO STATEMENTS WORK TOGETHER TO READ 3 VALUES
```

Ordinarily FORTRAN statements are coded on a form similar to the above. Each line is the equivalent of one punched card. Only columns 7 to 72 are used for FORTRAN coding.

Comment

A C in column one indicates that the line is a comment; the compiler is not to process it. A comment does not affect the program in any way.

FORTRAN statements may continue from one line to the next. The first line of a statement is termed the initial line. The following lines of this statement are termed continuation lines.

Column 6 of the coding form is used to distinguish between initial and continuation lines. An initial line contains either a 0 or a blank in column 6. A continuation line contains any character but a 0 or blank in column 6. Any one statement may be composed of one initial line and up to 19 continuation lines.

Statement Labels

Columns 1 to 5 of an initial line may contain a numeric statement label so that the statement may be referenced by other statements. The label may consist of from 1 to 5 digits. The label may be placed anywhere in columns 1 to 5. The same label may not be given to two or more statements. 0 is not a valid statement label.

End line

Each program unit coded, a main subprogram or dependent subprogram, must have one end line. This is the last line of the program unit.

An end line contains only the characters E, N and D in that order and blanks somewhere in columns 7 to 72.

Blanks

Blanks have no significance in most FORTRAN statements. Thus X=A+B and X = A + B are equivalent. The few exceptions are noted in the relevant discussions.

Constants, Variables, Arrays and Subscripts

All constants and variables fall into one of the following types: integer, real, double precision, complex, logical or Hollerith.

A constant is a datum which does not change during the execution of a program.

An integer constant is written as a string of decimal digits, optionally preceded by a plus or minus sign. If the sign is omitted plus is assumed.

Examples +5, 1768, -324

A real constant may be written in one of two ways.

1. An integer part, a decimal point and a fractional part. Both the integer and fractional part are strings of decimal digits. One of the strings may be omitted but not both. A sign may precede the constant, if it is omitted plus is assumed.

Examples 5.4, 25., -.37, +72

2. The basic constant above or an integer constant may be followed by the "E" and an optionally signed integer constant. This exponent means that the basic constant is to be raised to that power of 10 represented by the integer constant following the "E".

Examples	1.5E2	means	$1.5X10^2$
	-32E5		$-32.X10^5$
	.54E-7		$.54X\ 1/10^7$

A double precision constant is written similar to the type 2 real constant above except that the letter "D" is used instead of the letter "E". This indicates that the constant is to be stored in two storage units rather than one.

Examples	174.376D2	means	$174.376X10^2$
	-5.9 D-3		$-5.9X\ 1/10^3$

A complex constant is written as two optionally signed real constants, separated by a comma and enclosed within parentheses.

This value is stored in two storage units.

Examples	(6.5, 3.4)	means	$6.5X10^1+3.4iX10^1$
	(-7.E2, -37.)		$-7.X10^2\ -37.iX10^1$
	(.4E-2,.17)		$.4X1/10^2 + .17iX10^1$

There are only two logical constants written.TRUE. and .FALSE.

A hollerith constant is written as an integer constant (n), the letter "H" and exactly n characters. The n characters are the constant value. Any character legal in the particular processor being used may appear in a hollerith constant. Note that a blank is a legal character in such a constant.

5HABCDE
4H12 3
6HAB-468

A variable is a datum which may be changed during execution of a program. There are five types of variables integer, real, double precision logical and complex. Each variable used in a program must be assigned a name.

A variable name is composed of from one to six alphanumeric characters. The first character must be alphabetic; it implicitly defines the type of a variable: A-H and O-Z implies real type I-N implies integer.

Examples	Name	Implied Type
	A	real
	I	integer
	AIJ	real
	N52	integer

This implied typing can be overriden for integer and real variables and it must be overriden to define complex, logical and double precision variables. This is done with the data type statements (page 3-10)

An integer variable may contain only integral values. Its size is one storage unit.

A real variable may contain real values.
Its size is one storage unit. It consists of a mantissa and an exponent. When a value is stored in a real variable the decimal point is floated to the left of the first digit and the exponent is adjusted to reflect this shift. Thus real is also termed floating point.

Examples 1.5 is stored as .15X10^{2}
32.76E2 is stored as .3276X10^{4}
.012 is stored as .2X10^{-1}

A double precision variable may contain real values. Its size is two storage units. It consists of a mantissa and exponent. The extra storage unit is dedicated to the mantissa. Thus while a double precision variable does not store larger values than a real variable, it is capable of greater precision.

A logical variable may contain only the value true or false. Its size is one storage unit.

A complex variable may contain only complex values. Its size is two storage units, each with a mantissa part and an exponent part. In fact two real values are stored in a complex variable. The leftmost unit represents the real part of the value and the rightmost unit represents the imaginary part.

Example 3.5+7.4i is stored as

$.35X10^2$ and $.74X10^2$

An array is an ordered set of contiguous variables. An array may be any of the five types: logical, integer, real, double precision or complex. All of the entities in the array are of the same type.

Arrays are named and typed in the same way that non-contiguous variables are named and typed. The individual entities making up the table are termed elements.

The name assigned to an array is also applied to all of the elements in that array, thus a subscript (see below) must be used to indicate which element is being referenced in executable statements.

An array may be one, two or three dimensions. The number of dimensions and the size of each dimension in an array must be declared with a DIMENSION, COMMON or type statement (Section 3) prior to use in the program.

A one dimension array consists of one row of n elements

```
A(1), A(2),...,A(n-1), A(n)
```

A two dimension array consists of m rows of n elements each

```
A(1,1),A(1,2),...,A(1,n-1),A(1,n)
A(2,1),A(2,2),...,A(2,n-1),A(2,n)
  .                             .
  .                             .
  .                             .
  .                             .
A(m,1),A(m,2),...,A(m,n-1),A(m,n)
```

A three dimension array consists of k planes of mXn elements each.

```
A(1,1,1),...,A(1,n,1)  .
   .             .       .  A(1,1,k),...,A(1,n,k)
   .             .      .      .             .
   .             .     .       .             .
A(m,1,1),...,A(m,n,1)   .      .             .
                          .  A(m,1,k),...,A(m,n,k)
```

A subscript is the pointer used to make a reference to an array element unique.

It consists of one, two or three subscript expressions depending upon whether the array has one, two or three dimensions. The entire subscript is enclosed within parenthesis. Each subscript expression is separated from its successor by a comma.

A subscript expression may be an integer constant, an integer variable or a combination of these. The legal combinations are

```
v
k
v-k
v+k
c*v
c*v-k
c*v+k
```

Where v is a variable and c and k are constants

In a two expression subscript, the major expression defines row the minor expression defines column.

Example A(2,1) refers to that element at the intersection of row 2 and column 1.

In a three expression subscript, the major expression defines row, the middle expression defines column and the minor expression defines plane.

Example A(3,2,4) refers to that element at the intersection of row 3 and column 2 in the fourth plane.

SPECIFICATION STATEMENTS

Specification statements give information about storage requirements and about variables and constants used in a program. Specification statements must precede executable statements in a program unit.

The following statements will be discussed here

DIMENSION
COMMON
EQUIVALENCE
EXTERNAL
Type statements
DATA *

* Properly speaking, the DATA statement is not classed as a specification statement, but is a type unto itself. It is included here because it fits the above definition.

600

DIMENSION

DIMENSION $V_1(i_1\ ),\ V_2\ (i_2\ ),\ldots,\ V_n\ (i\ \ )$

where each v is an array name and each i is a declarator subscript of one, two or three expressions, separated by commas.

Dimension statements declare the names of arrays (page 2-3) used in the program unit and define the number of dimensions and size of each dimension. Arrays may have one, two or three dimensions.

Examples

Single Dimension Arrays

```
DIMENSION A(6), B (100)
```

defines two arrays A and B. A has six elements A (1) through A (6). B has 100 elements B (1) through B (100).

Two dimension arrays

```
DIMENSION C(2,3), D(10, 10)
```

defines two arrays C and D. C has six elements C(1,1) through C(2,3). D has 100 elements D(1,1) through D (10,10)

Three dimension array

```
DIMENSION E(3,4,5)
```

defines an array E with 60 elements E(1,1,1) through E(3,4,5)

Note that arrays may be declared in a COMMON statement and dxplicit data type statements. Such arrays may not also be declared in a DIMENSION statement.

An array name and its dimensions may be passed to a function or subroutine subprogram (section 9) in the list of arguments. In this Case a dummy array must be declared in the subprogram and its dimension declarators be given as integer variable rather than constants. The following example illustrates such adjustable dimensions.

```
in main subprogram
    DIMENSION ARAY (10. 10)

    CALL SUBSP (..., ARAY, 10, 10...)
```

595

in Subroutine subprogram

```
SUBROUTINE SUBSP (...,XRAY,I,J...)
                 ↓
DIMENSION  XRAY(I,J)
```

Array with adjustable dimensions may not be defined in main subprograms.

COMMON

There are two types of common storage ,blank common and labeled common. Each is defined by its own form of the COMMON statement.

Blank common

COMMON $a_1, a_2, \ldots a_n$

each a is a variable name, array name, or array name with subscript declarators.

the order in which storage is assigned to entities in blank common is determined by the sequence of the list or lists in the COMMON statement or statements.

Examples

```
1.    COMMON  A,B(5),R,I

2.    COMMON  A,B(5)

      COMMON  R,I
```

1 and 2 are equivalent.

Labeled common.

COMMON $/X_1/a_1, \ldots, a_n/X_2/a_1, \ldots, a_n$

each a is as defined for blank common. Each X is the name to be assigned to a block of common

These named blocks of common are collectively referred to as labeled common.

The order in which storage is assigned to entities in a block of labeled common is determined by the sequence of the list or lists in the COMMON statement or statements.

Example

```
COMMON /A/ A,B,C/B2/I,J

COMMON/A/ D
```

the order within block A is A,B,C,D

Both types of Common may be defined with the same COMMON statement as shown in this example.

```
COMMON A,B/B1/X(5,4),Y//C/B2/I,M
```

this defines in blank common A,B,C

in labeled common block B1 X(5,4),Y

in labeled common block B2 I,M

the purpose of common storage is to allow reference to the same variables and arrays by a main subprogram and its dependent function subprograms and subroutine subprograms (Section 9) To share common storage two subprograms must both define common with a COMMON statement.
A variable or array need not be given the same name in the two definitions because the correspondence is by relative position, not by name. If labeled common is being defined the block names in the two definitions must be the same.

example

in main subprogram

```
COMMON  A,X(4),B
```

in dependent subprogram

```
COMMON A,B,C,D,E,F
```

```
A     =A
X(1)  =B
X(2)  =C
X(3)  =D
X(4)  =E
B     =F
```

one use of labeled common is to make smaller the size of common any one dependent subprogram must define.

Suppose the main subprogram and its two dependent subprograms require six real variables in common, three for each of the dependent subprograms. Using blank common it could be written

in main subprogram

```
COMMON A,B,C,D,E,F
```

in dependent subprogram one

```
COMMON U,V,W,X,Y,Z
```

in dependent subprogram two

```
COMMON P,Q,R,S,T,U
```

But using labeled common

in main subprogram

```
COMMON /BLK1/A,B,C/BLK2/D,E,F
```

in dependent subprogram one

```
COMMON /BLK1/U,V,W
```

in dependent subprogram two

```
COMMON /BLK2/S,T,U
```

EQUIVALENCE

EQUIVALENCE $(a_1,a_2,\ldots,a_n),(b_1,b_2,\ldots,b_n)$..

each a and b is a variable or subscripted array element

It is the purpose of the EQUIVALENCE statement to assign two or more entity to the same storage unit. The EQUIVALENCE statement can not be used to mathematically equate two entities (A=B).

If a two storage unit entity (complex or double precision) is equivalenced with a one storage unit entity (integer, real or logical), the latter will share space with the first storage unit of the former.

example, not in common storage

```
DIMENSION X(4), Y(8)

EQUIVALENCE (X(3), Y(2) )

yields X(1) X(2) X(3) X(4)
            Y(1) Y(2) Y(3) Y(4) Y(5) Y(6) Y(7) Y(8)
```

entities defined in common storage may be equivalenced, but this may not be done in such a way that it would require extending common beyond the first entity defined

If two entities are directly defined into common they may not be equivalenced.

example of proper usage.

```
COMMON X(5)

DIMENSION Y(5)

EQUIVALENCE (X(4), Y(2))
```

Common storage must be extended beyond X(5) to include Y(4) and Y(5); this is all right.

examples of improper usage.

```
1.      COMMON X(5)

        DIMENSION Y(5)

        EQUIVALENCE (X(2), Y(4)
```

this would require extending common in front of X(1) to include Y(1) and Y(2).

```
2.      COMMON  A,B,C
        EQUIVALENCE  A,C
```

two entities in common may not be equivalenced.

EXTERNAL

EXTERNAL $p_1, p_2, \ldots, p_n$
each p is a procedure name

If an argument to a call for a function or subroutine subprogram (Section 9) is the name of another function or subroutine subprogram, this subprogram must be declared as external to the calling subprogram.

example

```
EXTERNAL  SUBA
     ↓
CALL SUBB (X,Y,SUBA)
   ↓
```

Type statements

There are five statements which maybe used to explicitly declare the type of data.

REAL a_1, b_1, c_1
INTEGER a_2, b_2, c_2
DOUBLE PRECISION a_3, b_3, c_3
COMPLEX a_4, b_4, c_4
LOGICAL a_5, b_5, c_5
each a, b and c is a variable name or an array name, optionally, including dimension information

These statements are used to override the type implied in the first character of a data name (A-H, O-Z REAL; I-N INTEGER)

Arrays may be dimensioned in a type statement or declared as to type in a type statement and dimensioned in a dimension statement.

examples

```
REAL IMAGE, JAX
COMPLEX C(10), CA
DOUBLE PRECISION  DP
DIMENSION DP(5,5)
```

DATA

DATA $V_1/k_1/,V_2/k_2/,\ldots, V_n/k_n/$
each V is a list of variables and array elements, each k is a list of constants to be associated with the preceding variables

It is the purpose of the DATA statement to assign initial values to variables and array elements. These values may be altered during execution.

Any constant may be preceded by a repeat factor in the form n* where n is the number of repetitions and * indicates repeat.

The constant list must agree in type and number with the variable list.

examples

```
DATA A,B,I/5.4,3.72,4/,X/92.7/

DATA C,D,E,F/39.5,2*0.0,7.36/
```

A Hollerith constant may be assigned to a variable.

```
DATA AM/6HABCDEF/
```

this example supposes one storage unit can contain six characters

The DATA statement by itself may not be used to initialize entities in Common storage (page 3-4) Blank Common may not be initialized; labeled common must be initialized in a BLOCK DATA subprogram. (Section 9)

Arithmetic Assignment Statements

The general form of an arithmetic assignment statement is v=e, where v is a variable and e is an arithmetic expression.

Execution of the arithmetic assignment statement consists of:

1. evaluate the expression e.
2. Place the result in variable v.

Arithmetic expressions consist of arithmetic elements (variables and constants) and the arithmetic operators:

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

examples
```
X+Y
X+Y/Z+5.4
E**Z-B
```

Mixed mode expressions are not allowed; that is, one may not attempt to include various of the data types in one expression. Thus

X+I

is not allowed as X is Real and I is integer. There are certain exceptions to this general rule however:

1. Any arithmetic type may be exponentiated by an integer and it retains its type, thus
   X**2 or X**I is allowed.
2. A real element may be combined with a double precision or complex element, the result is double precision or complex.
3. A real element may be exponentiated by a double precision element, the result is double precision
4. A double precision element may be exponentiated by a real element the result is double precision.

The evaluation of arithmetic expressions follows set rules.

1. The expression is scanned left to right and all exponentiation is resolved.
2. The expression is scanned again and all multiplication and division is resolved.
3. The expression is scanned a third time and all addition and subtraction is resolved.

examples

`X+Y/Z`

means $X+\frac{Y}{Z}$

not $\frac{X+Y}{Z}$

`X+Y**2+Z`

means $X+Y^2+Z$

not $(X+Y)^2+Z$

nor $X+Y^{2+Z}$

nor $(X+Y)^{2+Z}$

Parentheses may be included in an arithmetic expression to document or to change the order of evaluation. Everything within parentheses is completely resolved before any part of the remainder of the expression is resolved.

examples

$\frac{X+Y}{W+Z}$

could be written

`(X+Y)/(W+Z)`

$(X+Y)^2$

could be written

`(X+Y) **2`

Once the expression has been evaluated the resultant value is placed in V.
The = means then "is replaced by" not "equals". V need not be of the same data type as the expression: if it is not the result will be adjusted according to the following table.

Type of V	Type of E	Action to be taken
I	I	evaluate and assign unchanged
I	R	Evaluate,truncate fraction, convert to integer format and assign.
I	DP	evaluate,truncate fraction, convert to integer format and assign.
I	C	not allowed.
R	I	evaluate, convert to real format and assign
R	R	assign unchanged
R	DP	evaluate, truncate if necessary and assign
R	C	not allowed.
DP	I	evaluate, convert to DP format and assign
DP	R	convert to DP format, evaluate and assign
DP	DP	assign unchanged
C	I	not allowed
C	R	not allowed
C	DP	not allowed
C	C	assign unchanged

Logical Assignment Statements

The general form of a logical assignment statement is v = e, where v is a logical variable or array element and e is a logical expression.

Execution of the statement consists of:

1. Evaluation of the expression
2. Placement of the result (TRUE or FALSE) in v.

Logical Expressions

A logical expression may only have the value TRUE or FALSE. It is formed with logical operators and logical elements.

A logical element may be a:

logical constant
logical variable
logical array element
logical function reference
or it may be more complex, a relational expression

Relational Expressions.

A relational expression consists of two arithmetic expressions (Section 4) and a relational operator.

The relational operators are:

operator	meaning
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GT.	greater than
.GE.	greater than or equal to

Arithmetic expressions of the type complex may not be used in relational expressions.

Both arithmetic expressions may be integer, real or double precision or one may be real and the other double precision.

Examples of relational expressions

```
X .EQ. Y
I .GE. 5
X+Y .LT. SQRT(W)+5.4
```

In all cases TRUE or FALSE is determined.

The logical operators are:

operator	meaning
.NOT.	logical negation
.AND.	logical conjunction
.OR.	logical disjunction

It is the function of these operators to build more complex expressions out of the elements (constant, relational expressions etc).

.AND. and .OR. join the expressions preceding and succeeding them. .NOT. however is unary; it applies only to the element immediately succeeding it. If .NOT. is to apply to two or more elements they must be enclosed within parentheses.

Examples

L1 .AND. L2	if both L1 and L2 are TRUE result is TRUE, else FALSE
X .EQ. Y .OR. L1	if X is equal to Y or if L1 is TRUE result is TRUE, else FALSE
.NOT. L1 .AND. L2	if L1 is FALSE and L2 is TRUE result is TRUE, else FALSE

Evaluation of logical Expressions.

The evaluation of logical expressions proceeds according to the following steps, left to right:

1. Within parentheses according to following steps then outside of parentheses
2. Arithmetic expressions according to rules for arithmetic evaluation (page 4- 2)
3. Relational operators
4. Logical function references
5. .NOT.
6. .AND.
7. .OR.

Examples

```
X .LT. Z .AND. .NOT. L1 .OR. Q .EQ. SQRT(W)
```

the order of evaluation is

1. `SQRT(W)`
2. `X .LT. Z`
3. `Q .EQ. SQRT(W)`
4. `.NOT. L1`
5. `X .LT. Z .AND. .NOT. L1`
6. `X .LT. Z .AND. .NOT. L1 .OR. Q .EQ. SQRT(W)`

CONTROL STATEMENTS

There are eleven control statements discussed in this section:

GO TO
assigned GO TO
computed GO TO
arithmetic IF
logical IF
CALL
RETURN
CONTINUE
PAUSE
STOP
DO

The ASSIGN statement will also be considered in this section because of its relationship with the assigned GO TO statement.

Control statements allow the programmer to control the exectution sequence of the program.

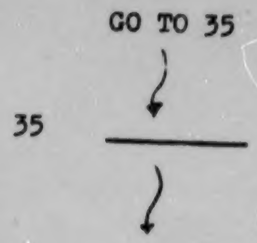
All statements referenced by control statements must be executable statements.

GO TO

```
GO TO  n
n is a statement label
```

Control is transferred to the named statement and execution continues at that point.

example



Assigned GO TO

GO TO i,($k_1,\ldots,k_n$)
i is an integer variable and each k is a statement label

ASSIGN statement

ASSIGN n TO i
i is an integer variable,n is an integer constant

Prior to executing the assigned GO TO, a legal value must be assigned to i by an ASSIGN statement. The only legal values are the k's in the assigned GO TO list. Execution of the assigned GO TO then causes transfer to the named statement.

example

```
      ASSIGN 5 TO J
      GO TO J(4,5,10,20)
    5 ASSIGN 20 TO J
```

Values may be assigned to i only by means of the ASSIGN statement.

Computed GO TO

GO TO (K_1,...., K_n),i
i is an integer variable, each K is a statement label

Prior to executing a computed GO TO a legal value must be assigned to i. This may be done with an arithmetic assignment statement (Section 4). At the time of execution of the computed GO TO statement i must be greater than zero and less than or equal to n. Execution of the computed GO TO causes transfer to the statement whose label is the i'th in the list.

Example:

```
J = 2

GO TO (4,5,10,20),J

5  J= 4
```

Arithmetic IF

> IF (e) k_1, k_2, k_3
> e is an arithmetic expression of any type, each k is a statement label.

Execution of the arithmetic IF will cause a transfer to k_1, k_2 or k_3 depending upon the arithmetic expression (Section 4), according to the following:

value of e	transfer to
<0	k_1
=0	k_2
>0	k_3

example IF (X-Y) 5,10,15

IF X-Y less than zero transfer to statement 5

IF X-Y equal to zero transfer to statement 10

IF X-Y greater than zero transfer to statement 15

All three k's must be written even in those cases programmer wishes to make the computed GO TO a one-way or two-way branch.

examples

```
   IF(X+4)  20,20,30
10________________
```

or

```
   IF (X+4) 20,20,30
20________________
```

Logical IF

IF (e) S
e is a logical expression,
S is true path

Execution of the logical IF causes the logical expression e (Secion 5) to be evaluated for True or False. If e is True, S is executed. If e is False S is bypassed and the next sequential statement is executed.

S may be any executable statement except a DO statement (page 6-12) or another logical IF

examples

```
IF(L)   GO TO 25
```

if the logical variable L is True transfer to statement 25.

```
IF (X.GE.Y.OR.SQRT(A).EQ.B)  Z=Z+1.0
```

add one to Z only if either X is greater than Y or the Square Root of A equals B.

CALL

CALL s $(a_1,\ldots,a_n)$

s is the name of a subroutine subprogram, each a is an actual argument

the CALL causes execution of the named subroutine subprogram (Section 9).

The arguments are passed to the subroutine for its use. The arguments must be of the proper number, order and type as required by the subroutine being called.

example

```
CALL SUBA (X,Y,I,5.4)
```

RETURN

The RETURN statement may only appear in function subprograms and subroutine subprograms (Section 9)

Execution of the RETURN in a subroutine causes control to be returned to the calling subprogram.

Execution of the RETURN in a function causes control to be returned to the calling subprogram and makes the value of the function available to it.

The RETURN is the only means by which control can be returned to the calling subprogram.

For example of usage see section 9.

CONTINUE

Execution of the CONTINUE causes no action; it is a means by which the programmer may insert a handy reference point where needed.

example

```
DO  10 I = 1,20,1
        ↓
IF (X.GT.Y)  GO TO 10
        ↓
10 CONTINUE
```

See DO statement (page 6-12)

STOP

STOP n
n is blank or a string of
one to five octal digits

Execution of the STOP causes program termination

The STOP may be coded wherever logically proper in a program. More than one STOP may be in a program, but only one will be executed in any one running of the program.

PAUSE

PAUSE n
n is blank or a
string of one to five
octal digits

Execution of a PAUSE causes the program to be temporarily suspended. Resumption of execution is not under program control. When the program is resumed the next sequential statement is executed with all values undisturbed by the suspension.

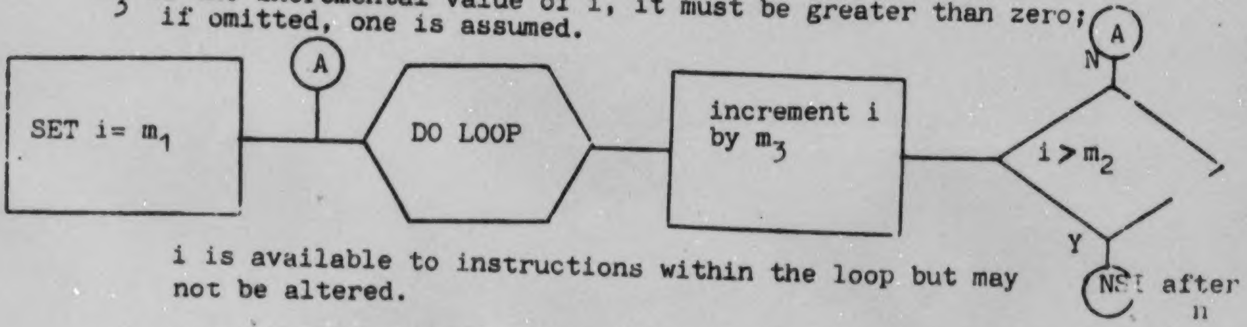
DO

DO n i = m_1, m_2, m_3
n is a statement label, i is an integer variable, each m is an integer constant or variable.

The DO statement controls a loop which consists of all statements beginning with the next sequential statement after the DO and ending with statement n.

Statement n must physically follow the DO statement and be in the same subprogram.

i controls the number of times the loop is executed
m_1 is the initial value of i, it must be greater than zero
m_2 is the limiting value of i, it must be greater than zero
m_3 is the incremental value of i, it must be greater than zero; if omitted, one is assumed.



i is available to instructions within the loop but may not be altered.

n may not be a GO TO, arithmetic IF, RETURN, STOP, PAUSE, DO or a logical IF which includes any of these.

example

```
        X=0.
        DO 20 I=1,10,1
  20    X = X + 1.0
```

at termination of this loop X will have 10.0 for its value.

One DO loop may have other loops within it, if they are completely nested

example

```
DIMENSION  X(10,10), Y(10,10)

DO 50 I = 1,10,1
DO 40 J = 1,10,1
X(J,I) = X(J,I) -Y(J,I)
40 CONTINUE
50 CONTINUE
```

It is also permissible to have both DO loops terminate on the same instruction; thus the above could also be written

```
          DO 50 I= 1,10,1
          DO 50 J= 1,10,1
      50 X(J,I) = X(J,I) - Y(J-I)
```

In a nest of DO loops, the innermost loop may have an extended range. This innermost loop may transfer out of the entire nest of DO loops with a GO TO or arithmetic IF. To continue execution as if the transfer had not occurred, a transfer is made back into the innermost loop with a GO TO or Arithmetic IF

example

```
          M = 0
          DO 10  I=J,K,L
          DO 20 II  =JJ,KK,LL

          GO TO 50
      30____________

      20 CONTINUE
      10 CONTINUE

          M = 1

      50_______________    }
                           }   extended
          IF (M-1) 30,60,60 }   range.
      60_______________
```

Input-Output Statements

There are two groups of Input-Output statements. The first group consists of the READ and WRITE; the statements which cause transmission of data between a central processor and peripheral devices.

The other group includes the ENDFILE, BACKSPACE and REWIND statements; the statements which manipulate data files.

Additionally the non-executable FORMAT statement can be used in conjunction with the READ and WRITE statements to control data format on the external medium.

There are two versions of the READ and WRITE statements, depending whether or not a FORMAT statement is used. Each will be discussed separately.

A READ or WRITE action in FORTRAN is not directed to a particular device such as a card-reader or printer. Rather it is directed to a FORTRAN unit number. Which numbers may be used as FORTRAN units is specific to a particular FORTRAN system. Also, how the relationship between a unit number and an actual hardware device is established is specific to a particular FORTRAN system. One example of how this might be done is one number is always associated with a particular device; for example the card-reader is unit 3 etc. Another way it might be handled is through Job Control Language. Each time a FORTRAN program is executed, unit numbers used in that program must be equated to devices. This method gives device independence to the programmer. In any case, the programmer must learn the method used by the FORTRAN system under which he is operating.

Non-formatted READ

READ (u) k
u is the unit number and k
is the I/O list

u may be an integer variable or constant.
k is a list of variables and array elements. Data read is placed in these list elements in the order specified by the list. The non-formatted READ rquires that the data appear on the external medium exactly as it is to appear in memory as no conversion will be performed.

The number of variables in the list may not be greater than the number which can be satisfied by one record. The unformatted READ will only read one record each time it is executed. If the list is absent from the READ a record will be read but will not be made available to the program. This, then, could be used to skip over selected records in a file.

Formatted READ

```
READ (u,f)k
 u is the unit number, f is
 a statement label or array
 name and k is the I/O list
```

u may be an integer constant or variable.

f is either the label of a FORMAT statement or an array name. This controls the conversion necessary to accomplish the READ.

k is the list of variables and array elements into which the data is to be read.

The formatted-READ requires that the data appear on the external medium in BCD format. The FORMAT statement controls the conversion of the data into the internal code of the processor.

The formatted-READ will read multiple records if necessary to satisfy the list.

If the list is omitted a record will be read into memory but will not be made available to the program.

Non-formatted WRITE

```
WRITE (u)K
 u is a unit number and
 k is an I/O list
```

u is an integer variable or constant.

k is a list of variables and array elements to be written to the external medium.

Data in the output record is not converted, it appears exactly as in memory.

Formatted-WRITE

```
WRITE (u,f)k
 u is a unit number, if a
 statement label or array name
 and k is an I/O list
```

u may be an integer constant or variable

f is either the label of a format statement or an array name. This controls the conversion necessary to accomplish the WRITE.

k is the list of variables and array elements to be written to the external medium.

The data will be converted to BCD format on the output medium.

FORMAT statement

The FORMAT statement is used with the READ and WRITE statements to provide conversion and editing information between the internal data representation and external BCD.

The FORMAT statement has the basic form:

n FORMAT ($fd_1,\ldots,fd_n$)

n is the statement label and each fd is a field descriptor

each field descriptor defines one data field in the external medium, it defines the field's width and the type of datum in the field.

The record is described left to right starting with the first position.

There are nine field descriptors: integer, real, real with exponent, double precision, general, logical, alphanumeric, hollerith and blank.

The general forms of the field descriptors are:

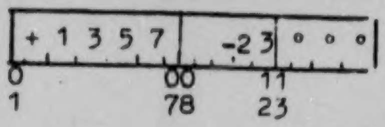
Iw	integer
Fw.d	real
Ew.d	real with exponent
Dw.d	double precision
Gw.d	general
Lw	logical
Aw	alphanumeric
nH	hollerith
nX	blank

I,F,E,D,G,L,A,H and X define the descriptor type.
W defines the number of positions in the field.
.d gives an assumed decimal point for input, it is overridden by an actual decimal point in the data. In output it defines the number of fractional positions wanted.

Integer descriptor Iw

Legal data characters are the numerics (0-9) and a sign (+ or -).
Input data must be right justified as imbedded and trailing blanks will be treated as zeros. When integer data are written the data will be right justified with blanks replacing leading zeros. A position must be allowed for a sign.

Example read two integer values.



```
     READ(2,12) J,K
13   FORMAT (I7,I5)
```

The value +13,570 is read into J under control of the I7 descriptor. The value-23 is read into K under control of the I5 descriptor.

The remainder of the record is not made available to the program.

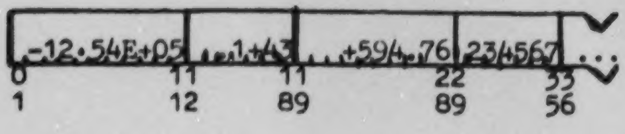
Real descriptors Fw.d, Ew.d, Gw.d

These three descriptors may be used to convert single precision real data.
In input these three descriptors are equivalent and any one may be used at the programmers discretion. Legal input consists of a basic value and an optional exponent.
Legal characters in the basic value are the numerics (0-9), a sign (+or-) and a decimal point.
The optional exponent may be of one of the following forms:

signed integer constant
E followed by an optionally signed integer constant.
D followed by an optionally signed integer constant.

Note that D in this case is overridden and the value is stored in one computer storage unit.

Example: read four real values



```
    READ (3, 13) X,4,Z,A
13  FORMAT (F11.0,F7.0,F10.0,F7.2)
```

All of these values could have also been converted using an E or G descriptor.

Because the first three values contain actual decimal points, the size given to d in the first three descriptors is ignored. The fourth value is read as +2345.67

There are differences between the F, D and G descriptors in output.

The F descriptor gives a right justified value with leading blanks, and a minus sign if negative. The fractional part of the value will be rounded to d places. The W in the descriptor should allow for a sign and must allow for a decimal point position.

Example:

internal value of X = $.123456.X10^2$
WRITE (5,16)X
16 FORMAT (F7.3)
gives output value 12.345

The E descriptor gives a general output form of $s0.n_1 \ldots n_d$ exp where:

s is a - if negative or no position if positive
$n_1 \ldots n_d$ are the d most significant digits of the value
exp is either E±nn or ±nnn depending on the size of the exponent value
0 may be no position in some FORTRAN systems

Example:

internal value of X=$.123456 \ldots X10^2$
WRITE (5,15)X
15 FORMAT (E10.5)
gives output value .12345E+02

The G descriptor in output can result in either a F or E type conversion. The determining factor as to which is used is the absolute magnatude of the value being written. Let M represent the magnatude of the value: if M is less than 10^d an F conversion will be used, if M is equal to or greater than 10^d an E conversion will be used.

M	Conversion used
$.1 \le M < 1$	Fw-4.d,4X
$1 \le M < 10$	Fw-4.d-1, 4X
$10 \le M < 100$	Fw-4.d-2, 4X
⋮	⋮
$10^{d-1} \le M < 10^d$	Fw-4.0, 4X
$10^d \le M$	sEw.d

s is a scale factor to be discussed on page 7-

Example

```
    WRITE (6,60) X
60 FORMAT (G9.3)
```

if X = .234 the conversion is F5.3,4X
which gives ▲.234▲▲▲▲

if X = 2.34 the conversion is F5.2,4X
which gives ▲2.34▲▲▲▲

if X = 23.4 the conversion is F5.1,4X
which gives ▲23.4▲▲▲▲

if X = 234. the conversion is F5.0,4X
which gives ▲234.▲▲▲▲

if X = 2340. the conversion is E9.3
which gives .2340+E04

Double precision descriptor Dw.d

This descriptor allows real values to be stored into two storage units rather than one and thus achieve greater precision. Legal input is the same as for the single precision descriptors E,F and G. Output is the same as for the single precision E descriptor except that D replaces E.

Complex conversions

There is no complex field descriptor; rather, because a complex datum is in fact two real data, a pair of real descriptors is used to effect conversion. The first of this pair defines the real part and the second defines the imaginary part.

Example:

```
          READ(3,32) C
       32 FORMAT (F7.2, F6.3)
```

Logical Descriptor Lw

For a value of True to be read into a logical variable the first non-blank character in the input field must be a T. For false the first non-blank character in the field must be an F. The contents of the remainder of the field are ignored. On output, the field will consist of W-1 blanks followed by a T or F depending upon the value of the logical variable.

Blank descriptor nX

On input n characters of the record will be skipped, they will not be stored into a variable. On output n blanks will be inserted into the record.

Hollerith descriptor $nHc_1...cn$

This descriptor is of use on output to define constant values which are to be placed in the record.
This descriptor is not associated with a variable in the I/O list, rather the n characters are transmitted.

Example:

```
          suppose X = 7.25

          WRITE (5,50) X
       50 FORMAT (14HTOTAL VALUE:   ,F6.2)

   gives  TOTAL VALUE:    7.25
```

Alphanumeric descriptor Aw

This descriptor causes w characters to be read into or written from one of the I/O list elements, which may be of any data type as there is no alphanumeric type variable. To achieve a proper conversion the programmer must know the number of characters which can be stored in one storage unit.

Suppose this number is c. On input if w is greater than c the rightmost c characters of the input field will be stored and the leftmost w-c characters will be lost to the program.

If w is less than c the w characters will be stored left justified with trailing blanks.

On output if w is greater than c the output field will consist of w-c blanks followed by the c characters. If w is less than c only the leftmost w characters will be written.

Scaling

A scale factor is defined for all F,E,G and D descriptors when format control begins a scale factor of zero is assigned. This can be altered at any point in the format statement by coding a scale factor with a field descriptor. It has the form nP where n is an unsigned or minus integer constant.

Example 3PF7.2

This new scale factor does not apply only to this descriptor but also to all F,E,G and D descriptors following, in this FORMAT statement, until another scale factor is defined.

The effect of the scale factor varies between the different descriptors and between input and output.

For input F,E,G and D descriptors, if the input data have no expressed exponents, the external value equals the internal value times 10 to the n power.

$$V_e = V_i X 10^n$$

Example

external value	descriptor	internal value
44.566	1PF7.3	4.4566

Thus a positive scale factor shifts the decimal point left n places, a negative scale factor shifts the decimal point right n places.

If the input data have expressed exponents the effect of the scale factor is negated for that data.

For F descriptors on output, again, the external value equals the internal value times 10 to the n power

$$V_e = V_i X10^n$$

This however gives an effect opposite to that for input.

Example:

Internal Value	Descriptor	External value
4.4566	1PF7.3	44.566

A positive scale factor shifts the decimal point right n places, a negative scale factor shifts the exponent left n places.

For E and D output the basic value is multiplied by 10^n and the exponent is reduced by n.

For G output, if the magnatude of the datum allows use of an F conversion the scale factor has no effect.
If an E conversion is used the effect is the same as for an E descriptor.

Repitition Constants

If a series of field descriptors are the same, instead of coding n separate descriptors, one description preceeded by the integer n may be coded.

Examples F7.2, F7.2, F7.2
may be written 3F7.2

I3, I3, I4, I4, I4
may be written 2I3, 3I4

Additionally a repitition constant may be applied to like groups of descriptors. The group is coded one time within parentheses with the repitition constant preceding the parenthesis.

Example I3, I4, I3, I4
could be written 2 (I3, I4)

One group may be imbedded within another.

Example

F7.2, F6.1, F8.2, F6.1, F8.2, F7.2, F6.1, F8.2, F6.1, F8.2

could be written 2(F7.2,2(F6.1, F8.2))

Scanning the FORMAT Statement.

The FORMAT statement is scanned left to right. The next field descriptor is associated with the next list element, except that two descriptors are associated with a complex element and no element is associated with a blank (nX) or hollerith descriptor ($nHc_1 \ldots c_n$).

The scan must continue until the I/O list is exhausted. If the end of the list is reached before the end of the FORMAT statement, the rest of the FORMAT statement is ignored. If the end of the FORMAT statement is encountered first however, a rescan of the FORMAT is begun. Anytime a rescan is necessary a new record is begun both on input and output.

With the simple FORMAT statement (no internal parentheses) the rescan starts at the first descriptor of the statement.

If there are internal parentheses however the rescan is started at that left parenthesis associated with the last internal right parenthesis in the statement.

Examples

```
FORMAT (I3, F15.5, F12.6, I7)
         ↑
         rescan point

FORMAT (I3, 2(F15.5, F12.6, I7))
            ↑
            rescan point

FORMAT (I4, 2(F15.2,3(I7,I5), F9.3))
            ↑
            rescan point

FORMAT (2(F7.2),2(I6,3(I7)).(F9.3))
                            ↑
                            rescan point
```

Print formatting.

When writing to the printer, the output record must contain printer vertical format control. This is supplied in the first position of the record. This position will be blanked out when the record is printed.

Legal vertical spacing characters are shown below.

Character	Space before Printing
Blank	One line
0	Two lines
1	Head of form
+	Do not advance

One method of inserting an advancing character into the print line would be to code a hollerith constant.

Example

```
         WRITE (3,45) X, Y, Z
      45 FORMAT (1H1, 4HX^=^,F9.3,4HY^=^,F9.3,4HZ^=^,F9.3)
```

Multiple record format

As stated previously, anytime a rescan of a FORMAT statement is necessary a new record is begun. The programmer can also force a new record at any point in the READ or WRITE operation by including the slash(/) character in the format statement.

```
Examples     READ(2,12) X,Y,Z
          12 FORMAT (F12.4, F10.3/F11.5)
```

X and Y would be read from the first record and Z from the second.

```
             READ (2,13) X,Y,Z
          13 FORMAT (F12.4,F10.3//F11.5)
```

X and Y would be read from the first record, Z from the third record. Record two would be skipped.

Short list notations

If an array name appears in an I/O list without subscript, this is taken to mean that every element of that array is to have a value read into or written from it.

```
Example      DIMENSION      A(4)

             READ (3,33) X,A,Y
          33 FORMAT (F7.1, 4F10.2, F9.2)
```

The order in which the list is satisfied is

X,A(1), A(2), A(3), A(4), Y

Formatting at execution time.

A READ or WRITE need not get its formatting information from a FORMAT statement. This information can be in an array.

Example READ (3,ARRAY1) A,B,C,D

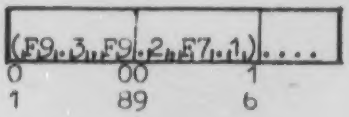
The information can be placed into the array by one of two ways. The array can be initialized with a DATA statement. The alternative is to read the format information into the array using an alphanumeric descriptor. The input record or records must contain everything just as it would be coded in a FORMAT statement with the exception that the word FORMAT is not included.

Example

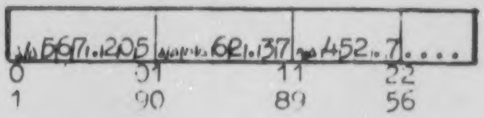
Suppose one storage unit can contain eight characters

```
DIMENSION A(2)

 5 READ (2,10) A
10 FORMAT (2A8)
15 READ (2,A) X,Y,Z
```

The data read by statement 5 could be:



Thus, this format information would control statement 15 which would read the data. An example of such data is:



DO implied lists

In addition to the simple I/O list and the short list discussed previously, a READ or WRITE statement may also contain a DO implied list.

The form of a DO implied list is:

($V_1,\ldots,V_n,i=m_1,m_2,m_3$)
each V is an array name subscripted
by i or a variable

The order in which the array elements are processed is controlled by the DO loop.

Examples:

```
      DIMENSION A(20)

      READ (3,37)(A(J),J=1,10)
   37 FORMAT (10F7.2)
```

Values are read into the first 10 elements of array A.

```
      DIMENSION X(10)

      WRITE (5,50)Y(Z,X(K),K=1,10)
   50 FORMAT (F15.5,(F12.2,F9.3))
```

The order in which the elements are written is

Y,Z,X(1),Z,X(2),Z,X(3),Z,X(4),Z,X(5),Z,X(6),Z,
X(7), Z, X(8), Z,X(9),Z,X(10)

One DO implied list may be imbedded within another.

Examples:

```
      DIMENSION    X(5,4)

      READ    (3,90)((X(I,J),I=1,5),J=1,4)
   90 FORMAT  (20F7.2)
```

The order in which the elements are processed is:

X(1,1), X(2,1), X(3,1), X(4,1), X(5,1), X(1,2),...,X(5,4)

Auxiliary Input-Output Statements

The three statements REWIND, BACKSPACE and ENDFILE are used to manipulate data files.

REWIND

The REWIND statement has the form

REWIND u

This statement causes unit u to be positioned at its starting point.

BACKSPACE

The BACKSPACE statement has the form

BACKSPACE u

This statement causes unit u to be repositioned such that the last record referenced will also be the next record referenced when a READ or WRITE is issued. This statement has no effect if u is at its initial point.

ENDFILE

The ENDFILE statement has the form

ENDFILE u

This statement causes a unique end of file record to be written on file u. Action is not defined if an end-of-file record is encountered during a read operation.

STATEMENT FUNCTIONS

The purpose of statement functions is to eliminate repetitive coding of the same expression in a subprogram.

For example if an expression of the form $V_1 + V_2 / V_3$ appears in ten statements in a subprogram, it might be advantageous to code the expression one time as a statement function and include a reference to it in the ten statements.

The general form of a statement function is:

$f(a_1, \ldots, a_n) = e$

f is the function name, each a is a dummy argument to the function and e is an arithmetic or logical expression

The name assigned to the statement function must be unique in the subprogram in which it is coded.

The rules for naming statement functions all the same as for variables and arrays

1. one to six alphanumeric characters.
2. the first must be alphabetic or it implies real (A - H, O - Z) or integer (I-N) type
3. to override 2 a type statement must be included in the subprogram to explicitly type the statement function.

The dummy arguments to the statement function need not have names unique in the subprogram as they do not define actual entities.

The expression must agree in type with the name assigned to the function (e.g. if the function is typed logical, the expression must be logical).

The expression may contain

the dummy arguments
non-hollerith constants
variable references
references to other statement functions defined previously
Intrinsic and external function references (Section 9)

All statement functions in a subprogram must be coded prior to all executable statements.

A statement function can be used only in the subprogram in which it is coded.

Thus if the various subprograms of a program all require a particular statement function, it would have to be coded in each.

A statement function is executed by including a reference to it in an executable statement, including real arguments of the proper type, order and number.

examples of usage

```
                  ↓
        SFUN (A,B) = SQRT (A**2+B**2)
                  ↓
   10   X =  Y + SFUN (Q,R)/Z
                  ↓
   20   B = SFUN (C,D) * E+F
                  ↓
```

Subprograms

There are three types of dependent subprograms which the programmer may code to use in conjunction with the main subprogram: the Function Subprogram, the Subroutine Subprogram and the Block Data Subprogram.

The first two of these would contain commonly used routines. Their usefulness is that a routine could be written one time and included in each program as required, unaltered, thus saving some amount of coding.

The block data subprogram does not contain executable code; its function is to initialize blocks of labeled common.

Block Data Subprogram

Blocks of labeled common (page 3-4) may only be initialized through a block data subprogram.

The first statement in a block data subprogram must be BLOCK DATA.

The last statement must be END.

The only statements which may appear within a block data subprogram are: the data type statements, EQUIVALENCE, DATA, DIMENSION, and COMMON

If one entity in a block of labeled common is initialized, the entire block must be described.

Example

in main subprogram

```
COMMON  /BLK1/ A,B,C

END

BLOCK DATA

COMMON  /BLK1/ X,Y,Z

DATA  X,Z/6.97, 17.64/

END
```

Note that variable Y(which is also variable B) is not initialized. It would be incorrect however to have a common statement such as

```
COMMON  /BLK1/ X,Z
```

The entire block must be defined.

The block data subprogram may not be used to initialize blank common.

Function Subprograms

The first statement of a function subprogram is of the form:

```
t FUNCTION f (a1,...,an)
      t is the function type, f is
      the function name and each a
      is a dummy argument
```

Function subprograms are given a type (e.g. INTEGER) or if omitted, the type is implied in the first character of the function name: A-H and O-Z real; I-N integer. The name must be one to six alphanumeric characters with the first character being alphabetic.

The dummy arguments may not appear in an EQUIVALENCE, DATA or COMMON statement.

The last statement of the function subprogram must be an END statement.

A function subprogram may return any number of values to the calling subprogram. This is done in two ways.

The name of the function itself must appear in at least one executable statement, to the left of the "=".

Also the function subprogram may redefine any of the dummy arguments and thereby redefine the true arguments of the call to the function.

Return to the calling subprogram is made by a RETURN statement (page 6-8). Return is made to the statement which issued the call to the function.

Calling function subprograms. A function subprogram is executed by including its name in an arithmetic or logical assignment statement. The true arguments must agree with the dummy arguments in type, number and order.

True arguments may be one of the following:

variable

array

array element

an external procedure (see EXTERNAL page 3-9)

an expression

Example

in main subprogram

```
     AREA=FIGR(WDTH,HGHT,J)
     IF (J.GT.0) GO TO . . .

     END
     FUNCTION FIGR(A,B,I)
     IF (A.  LE. 0.0) GO TO 42
     IF (B.  LE. 0.0) GO TO 44
     I = 0
     FIGR = A*B
     RETURN
13   FORMAT (1H0, 11H NEG WIDTH:, F7.2)
42   WRITE (5, 13) A
     I = 1
     FIGR = 0.0
     RETURN
44   WRITE (5,14) B
14   FORMAT (1H0, 12H NEG HEIGHT:, F7.2)
     I = 2
     FIGR = 0.0
     RETURN
     END
```

Subroutine Subprograms

The first statement of a subroutine subprogram is of the form:

SUBROUTINE s ($a_1,\ldots,a_n$)
s is the subroutine name and each a is a dummy argument.

The last statement of the subroutine subprogram must be an END statement.

The dummy argumenst may represent variable names, array names or external procedures. These dummy arguments may not appear in DATA, COMMON or EQUIVALENCE statements.

A subroutine subprogram returns values back to the calling subprogram by redefining the dummy arguments and thereby the associated true arguments.

Return is made to the calling subprogram by a RETURN statement (page 6-7)

The true arguments to the call may be:

hollerith constant

variable

array

array element

an expression

external procedure

Example (same as for Function subprogram to show similarities and differences)

in main subprogram

```
CALL FIGR (WDTH, HGHT, J, AREA)

IF (J  .GT. 0) GO TO. . .

END

SUBROUTINE FIGR (A,B,I,C)

IF (A.  LE. 0.0) GO TO 42

IF (B.  LE. 0.0) GO TO 44
```

```
   I = 0
   C = A*B
   RETURN
13 FORMAT (1H0,11H NEG WIDTH:,F7.2)
42 WRITE (5,13) A
   I = 1
   C = 0.0
   RETURN
44 WRITE (5,14) B
14 FORMAT (1H0, 12H NEG HEIGHT:,F7.2)
   I = 2
   C = 0.0
   RETURN
   END
```

Predefined Functions.

There are two groups of functions, Intrinsic and Basic External, which must be supplied with the Fortran compiler.

The tables below list the two types of functions

Basic External Functions

			Type of	
Function	Meaning	Name	Argument	Function
Exponential	e^a	EXP	Real	real
		DEXP	d.p.	d.p.
		CEXP	complex	complex
Natural Logarithm	$\log_e(a)$	ALOG	real	real
		DLOG	d.p.	d.p.
		CLOG	complex	complex
Common Logarithm	$\log_{10}(a)$	ALOG10	real	real
		DLOG10	d.p.	d.p.
Trigonometric Sine	$\sin(a)$	SIN	real	real
		DSIN	d.p.	d.p.
		CSIN	complex	complex
Trigonometric Cosine	$\cos(a)$	COS	real	real
		DCOS	d.p.	d.p.
		CCOS	complex	complex
Hyperbolic Tangent	$\tanh(a)$	TANH	real	real
Square Root	$(a)^{1/2}$	SQRT	real	real
		DSQRT	d.p.	d.p.
		CSQRT	complex	complex
Arctangent	$\arctan(a)$	ATAN	real	real
		DATAN	d.p.	d.p.
	$\arctan(a_1/a_2)$	ATAN2	real	real
		DATAN2	d.p.	d.p.
Remaindering	$a_1 (\mathrm{mod}\ a_2)$	DMOD	d.p.	d.p.
Modulus		CABS	complex	real

Intrinsic Functions

Function	Meaning	Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	ABS IABS DABS	real integer d.p.	real integer d.p.
Truncation	sign of a times largest integer $\leq \|a\|$	AINT INT IDINT	real real d.p.	real integer integer
Remaindering	$a(\mathrm{mod}\ a_2)$	AMOD MOD	real integer	real integer
Largest Value	$\mathrm{Max}(a_1,..a_n)$	AMAX0 AMAX1 MAX0 MAX1 DMAX1	integer real integer real d.p.	real real integer integer d.p.
Smallest Value	$\mathrm{Min}(a_1,..a_n)$	AMIN0 AMIN1 MIN0 MINI DMINI	integer real integer real d.p.	real real integer integer d.p.
Float	convert integer to Real	FLOAT	integer	real
Fix	convert real to integer	IFIX	real	integer
Sign Transfer	sign of a_2 times/a_1/	SIGN ISIGN DSIGN	real integer d.p.	real integer d.p.
Positive Difference	$a_1-\mathrm{MIN}(a_1,a_2)$	DIM IDIM	real integer	real integer
Most significant part of d.p. argument		SNGL	d.p.	real
Real part of complex argument		REAL	complex	real
Imaginary part of complex argument		AIMAG	complex	real
Single to double precision		DBLE	real	d.p.

Intrinsic Functions

Function	Meaning	Name	Type of Argument	Type of Function
Express two real arguments in complex form	$a_1+a_2\ -1$	CMPLX	real	complex
conjugate of complex		CONJG	complex	complex

Both the basic external functions and the intrinsic functions are executed by including a reference in an executable statement.

The various functions will be discussed individually below.

Absolute value

ABS returns the absolute value of the real argument.

example X = ABS(Y)
if Y = -5.4, X = 5.4

IABS and DABS for integer and double precision value.

Truncation

AINT truncates fractional part of real argument and return this value in real format.

example X = AINT(Y)
if Y = 5.4, X = 5.0

INT similar to AINT except value returned is in integer format

example I = INT(Y)
if Y = 5.4, I = 5

IDINT same as INT for double precision arguments.

example I = IDINT(D)
if D = 5.4, I = 5

Remaindering

AMOD first real argument is divided by second real argument. True remainder is returned with sign of first argument.

example Y = AMOD(Y,Z)
if Y = 7.2 and Z = 1.4, X = 2.0

MOD same as AMOD for integer values

DMOD same as AMOD for double precision values.

Choosing Largest Value

AMAX1 determines which of the two or more real arguments in greatest and returns this value in real format.

Example X= AMAX1(A,B,C)
if A = 1.5, B = -9.4 and C=7.3; X=7.3

MAX1 determines which of the real arguments is greatest and returns the value in integer format.

AMAX0 determines which of the integer arguments is greatest as returns this value in real format.

Example X = AMAX0 (I,J,K)
if I =3, J = -5, K=4; X =4.0

MAX0 determines which of the integer arguments is greatest and returns this value in integer format.

DMAX1 determines which of the double precision arguments is greatest and returns this value in double precision format.

Choosing Smallest Value

AMIN1 , AMIN0 etc similar to the corresponding max function, except that the smallest value is returned.

Example

X=AMIN1 (A,B,C)
if A=1., B=5.4 and C=.9; X= .9

Conversion from integer to real

FLOAT converts an integer argument to real format, it is of use within expression which would otherwise be mixed mode.

Example

X = A+FLOAT(I)
X = A+I is illegal

Conversion from real to integer

IFIX converts a real argument to integer format. It is of use within expressions which would otherwise be mixed mode.

Example

I = J+IFIX(X)
I = J+X is illegal

Sign transfer

SIGN combines the sign of the second real argument and the absolute values of the first real argument and returns this value

Example
X = SIGN(Y,Z)
if Y=7.9 and Z = -1.23; X = -7.9

ISIGN similar for integer arguments

DSIGN similar for double precision arguments

Positive difference

DIM subtracts the second real argument from the first
If result is zero or positive this value is returned, if negative zero is returned.

Examples
X = DIM(Y,Z)
if Y = 5.0 and Z =3.7; X =1.3
if Y = 3.7 and Z = 5.0; X = 0.0
if Y =-5 and Z = -6.3; X = 1.3

IDIM similar for integer arguments.

Obtain most significant part of d.p. argument

SNGL truncates those low order digitsof the double precision argument which would not fit in one storage unit.

Obtain real part of Complex argument

REAL extracts the real half of a complex number

example
X=REAL(C)
if C = 5.4+3.7i; X=5.4

Obtain imaginary part of complex argument

AIMAG extracts the imaginary part of a complex number and returns this value as a real number.

example
X=AIMAG(C)
if C=5.4+3.7i; X=3.7

Convert to double precision format

DBLE converts a real argument to double precision format

Convert to complex format

CMPLX returns the first real argument as the real part and the second real argument as the imaginary part of a complex number

example
C = CMPLX (A,B)
if A = 79.5 and B = 34.62; C =79.5 + 34.62i

Conjugate complex

CONJG converts a complex argument to its conjugate form.

Example C1 = CONJG(C2)
if C2 = 5.9 + 2.71i, C1=5.9 -2.71i

Exponentiation

EXP raises the value e (2.71828...) to the power indicated by the real argument.

example X=EXP(Y)
if Y =2.0; X =e^2 or 7.389. . .

DEXP for double precision arguments

CEXP for complex arguments

Natural Logarithm

ALOG returns the logarithm to the base e of the real argument

example X = ALOG(Y)
if Y = 3.7; X = 1.30833

DLOG for double precision arguments

CLOG for complex arguments

Common logarithm

ALOG10 returns the logarithm to the base 10 of the real argument

example X=ALOG10 (Y)
if Y = 3.7; X =0.5682

DLOG10 for double precision arguments.

Trigonometric Sine

SIN returns the sine of the real argument. The argument must be expressed in radians.
1°≅.01745 radian, 1 radian≅57° 44', $\frac{II}{2}$ radians = 90°
example X = SIN (Y)
if Y = $\frac{II}{4}$ (45°) , X = .70718

DSIN for double precision arguments

CSIN for complex arguments

Trigonometric cosine

COS returns the cosine of the real argument expressed in radians.

Example X=COS(Y)
if Y = .59 (34°), X = .829

DCOS for double precision arguments

CCOS for complex arguments

Hyperbolic Tangent

TANH returns the hyperbolic tangent of the real argument in radians.

Square root

SQRT returns the square root of the real argument

Example: X = SQRT(Y)
if Y = 8.75; X = 2.958. . .

DSQRT for double precision arguments

CSQRT for complex arguments

Complex absolute value

CABS returns the absolute value in real format of the complex argument using the formula

$$(r^2+i^2)^{1/2}$$

Example X = CABS(C)
if C = 5.4-3.7i, X=6.5448

Arctangent

ATAN returns the Arctangent of a real argument

DATAN similar for double precision arguments

ATAN2 computes arctangent for two real arguments using formula A_1/A_2

DATAN2 similar for double precision arguments

INDEX

A

Adjustable dimensions 3-2
Alphanumeric field descriptor 7-8
Argument, Dummy 8-1, 9-3, 9-5
Arithmetic Assignment Statement 4-1
Arithmetic expression 4-1
Arithmetic IF 6-5
Arithmetic Operator 4-1
Array 2-3
Array names in I/O lists 7-11
ASSIGN 6-3
Assigned GO TO 6-3
Assignment statements
    Arithmetic 4-1
    Logical 5-1

B

BACKSPACE 7-14
Basic external function 10-1
Blank Common 3-4
Blank descriptor 7-7
Blanks in statement 1-2
BLOCK DATA subprogram 9-2
Block of labeled common 9-2

C

CALL 6-7
Character set 1-1
Coding form 1-2
Comments 1-2
COMMON
  Blank 3-4
  Labeled 3-4
COMPLEX 3-10
Complex Constant 2-1
Complex variable 2-3
Computed GO TO 6-4
Constant 2-1
CONTINUE 6-9
Continuation of a statement 1-2
Control statement 6-1

D

DATA 3-11
Data type statements 3-10
Descriptors
  Aw 7-8
  Dw.d 7-6
  Ew.d 7-4
  Fw.d 7-4
  Gw.d 7-4
  nH 1-7
  Iw 7-4
  Lw 7-7
  nX 7-7

DIMENSION 3-2

DO 6-12

DO loop, nested 6-13

DOUBLE PRECISION 3-10

Double precision Constant 2-1

Double precision variable 2-2

E

END 1-2

ENDFILE 7-14

EQUIVALENCE 3-7

Expression

  Arithmetic 4-1

  Logical 5-1

  Relational 5-1

Extended range of do loop 6-13

EXTERNAL 3-9

F

Floating point 2-2

FORMAT

  Multiple Record 7-11

  Read in at execution 7-12

FORTRAN unit numbers 7-1

Function, basic external 10-1

Function, Intrinsic 10-1

Function, statement 8-1

Function Subprogram 9-3

G

GO TO 6-2
GO TO, Arithmetic 6-3
GO TO, Computed 6-4
Group repetition constant 7-9

H

Hollerith Constant 2-2
Hollerith Field descriptor 7-7

I

IF, Arithmetic 6-5
IF, logical 6-6
Imaginary part of complex number 2-3
Implied DO 7-12
Input
    Formatted READ 7-2
    Non-Formatted READ 7-1
INTEGER 3-10
Integer Constant 2-1
Integer variable 2-2
Intrinsic function 10-1

L

Label, statement 1-2
Labeled common 3-4
LOGICAL 3-10
Logical Assignment statement 5-1
Logical Constant 2-1

Logical IF 6-6
Logical operator 5-1
Logical variable 2-3

M

Mixed mode Arithmetic 4-1

N

Names of
  Arrays 2-3
  Subprograms 9-3
  Variables 2-2

O

Operators
  Arithmetic 4-1
  Logical 5-2
  Relational 5-1
Output
  Formatted WRITE 7-2
  non-Formatted WRITE 7-2

P

Parentheses
  in Arithmetic expressions 4-2
  in FORMAT statements 7-10
PAUSE 6-11
Printer vertical formatting 7-10

R

REAL 3-10
Real Constant 2-1
Real part of Complex number 2-3
Real variable 2-2
Relational expression 5-1
Relational operator 5-1
Repetition Constant 7-9
RETURN 6-8
REWIND 7-14

S

Specification Statements 3-1
    COMMON 3-4
    data type 3-10
    DIMENSION 3-2
    EQUIVALENCE 3-7
    EXTERNAL 3-9
Statement function 8-1
STOP 6-10
Subprograms
    BLOCK DATA 9-2
    Function 9-3
    Subroutine 9-5
Subscript 2-4

T

Type Statements 3-10

U

Unit numbers, FORTRAN 7-1

V

Variables 2-2

Vertical formatting, Printer 7-10

STUDENT HANDOUT
E3ABR51131 001
KDA-292

Technical Training

Programming Specialist

EXAMPLES OF STRUCTURED CODE

April 1978

USAF TECHNICAL TRAINING SCHOOL
3390th Technical Training Group
Keesler Air Force Base, Mississippi

Designed For ATC Course Use

DO NOT USE ON THE JOB

General Comments

Although ANS FORTRAN does not support the basic structured programming figures, it accommodates simulation of these figures. However, problems will be encountered in simulating the structured figures. First, there is no automated verification of the correctness of the simulated figures, and second, the branching which must be used in these figures adds complexity during the program coding and checkout phases. In addition, the lack of an INCLUDE capability, and FORTRAN's lack of a block structure, hamper top down programming.

The logical IF (as distinguished from the arithmetic IF) statement in FORTRAN does not provide the IFTHENELSE capability, i.e., two possible paths, but rather allows conditional execution of only one statement provided the logical expression tested is true. The IFTHENELSE figure thus must be simulated with a logical IF and one or more GOTO statements.

The DO statement in FORTRAN is an indexing type of looping statement which is of the DOUNTIL form since the test occurs at the end of the loop. The capability of iteration based on whether a condition is true or false does not exist. Therefore the DOUNTIL and DOWHILE control figures must be simulated with a logical IF and GOTO statements.

FORTRAN has no implementation of the CASE figure as such. However, use of the computed GOTO along with the unconditional GOTO statements provides a functional basis for the simulation of the CASE figure.

The following subsections detail a simulation of the basic structured programming figures and provide a set of suggested language coding conventions which are intended to have a minimal impact on the ANS FORTRAN user. They are all intended to achieve basic goals: to produce programs which are easy to write and debug, easy to read and understand, and easy to maintain and modify.

In summary the deficiencies in FORTRAN which affect its top down structured programming capability are:

a. The lack of an INCLUDE capability.

b. The lack of both the DOWHILE and DOUNTIL control structures.

c. The lack of the IFTHENELSE control structure.

d. The lack of any delimiters to facilitate block structuring.

Top Down Structured Programming in ANS FORTRAN

As mentioned previously, a basic problem in simulating the structured programming figures using the features of ANS FORTRAN lies in the fact that there is no automated verification of the structured figure's integrity. Thus, an incorrectly coded structured figure might appear correct in form. No solution to this problem is proposed for ANS FORTRAN figure simulation. A precompiler can eliminate this problem.

In defining each standard figure in this section, the goal has not been to attempt to minimize the number of lines written; rather it has been to strive for clarity so that the identity of each figure can be understood without any doubt. As an example of the emphasis on clarity stressed in these recommendations, the DOWHILE as suggested in the "IBM FSD Structured Programming Guide" (8):

```
s1        CONTINUE
          IF  (.NOT.(logical expression)) GO TO s2
            statements to execute
            GO TO s1
s2        CONTINUE
```

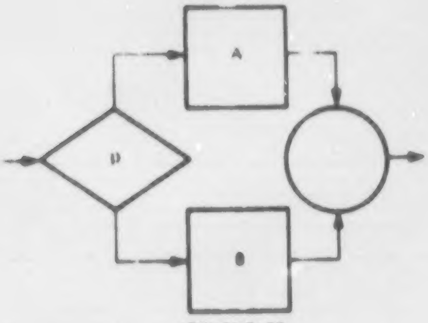
may be compared to the proposed DOWHILE simulation:

```
C         DO WHILE
            GO TO s2
s1          statements to execute
s2          IF (logical expression) GO TO s1
C         ENDDO
```

Note that comments are used to explicitly name the beginning and end of the DOWHILE in the latter.

IFTHENELSE Figure

The IFTHENELSE figure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p). Since the logical IF statement in FORTRAN allows conditional execution of a single statement, provided the logical expression tested is true, the IFTHENELSE figure is simulated in FORTRAN using both a logical IF statement and GOTO statements. The flowchart for the IFTHENELSE figure is:



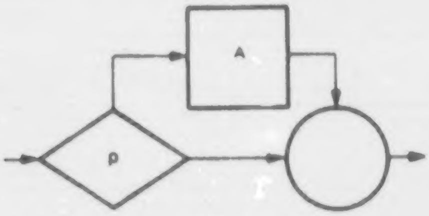
IFTHENELSE

The IFTHENELSE figure should be constructed so that the "true" conditional code (A) immediately follows the IF statement. In order to implement this in FORTRAN, it is more desirable to test for the negative of the condition desired by using a ".NOT." (a positive test would necessitate the use of an additional GOTO statement). The code structure recommended to represent the IFTHENELSE figure is:

```
         IF (.NOT.(p)) GO TO 0010
           code A
           GO TO 0020
C        ELSE
 0010      code B
 0020      CONTINUE
C        ENDIF
```

Statements within the two clauses including the GOTO terminating the first clause and the CONTINUE statement terminator for the figure should be indented two columns. The ELSE and ENDIF comment lines which aid in locating the end of the functional blocks of code should be aligned with the IF.

The ELSE in the IFTHENELSE figure is optional and if not used, the flowchart for this figure would be reduced to:



IFTHEN

and the code structure recommended to represent this is:

```
         IF (.NOT.(p))GO TO 0010
           code A
 0010      CONTINUE
C        ENDIF
```

If code A consists of a single statement the logical IF statement, which is a part of the FORTRAN language, may be used. The recommended format is as follows:

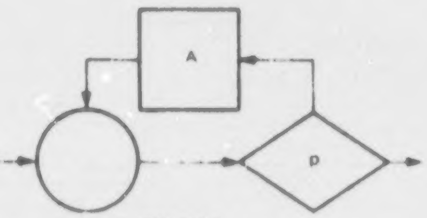
```
         IF (p)
     1     code A
C        ENDIF
```

DO Figures

The DO figures allow iterative execution of a functional block of code (A) based on a logical expression (p). If the test is made prior to the execution of code A it is a DOWHILE figure. If it is made after code A it is a DOUNTIL figure. The FORTRAN DO is essentially a specialized DOUNTIL and its use to simulate the DO figures is very clumsy. With the FORTRAN DO, execution continues as long as an index is not incremented past a test value; however, the DO statement is a command to execute, at least once, the statements within its range. The DO figures can easily and more understandably be simulated in FORTRAN using a logical IF statement and GOTO statements.

DOWHILE

It is recommended that the DOWHILE figure be simulated in FORTRAN using a logical IF statement and GOTO statements. The flowchart for the DOWHILE figure is:



DOWHILE

654 1,2

Note that the logical expression (p) is tested prior to each execution of the functional block of code (A) including the first.

Although this figure can be coded using either a positive conditional test or a negative conditional test, the positive conditional test approach is recommended for FORTRAN implementation. The code structure recommended to represent the DOWHILE figure is:

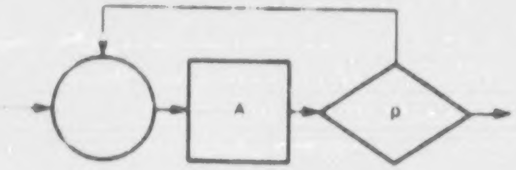
```
C        DO WHILE (p)
           GO TO 0020
 0010      code A
 0020      IF (p) GO TO 0010
C        ENDDO
```

The logical expression (p) on the DOWHILE comment line is recommended but could be deleted at the user's option.

Statements within the figure should be indented two columns from the DOWHILE and ENDDO comment lines which aid in locating the beginning and end of the figure.

DOUNTIL

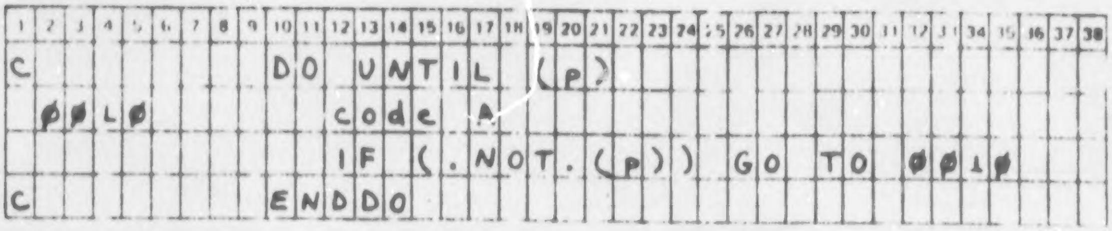
It is recommended that the DOUNTIL figure also be simulated in FORTRAN using a logical IF statement and a GOTO statement. When looping under control of an index, however, a FORTRAN DO might be an appropriate choice (refer to the next subsection). The flowchart for the DOUNTIL Figure is:



DOUNTIL

Note that the logical expression (p) is tested after each execution of the functional block of code, so that code (A) is always executed at least once.

The recommended simulation of this figure requires that the conditional test on the looping variable be negated as indicated in the example below. The negative condition is achieved by applying a ".NOT." to the desired logical expression. The code structure recommended to represent the DOUNTIL figure is:



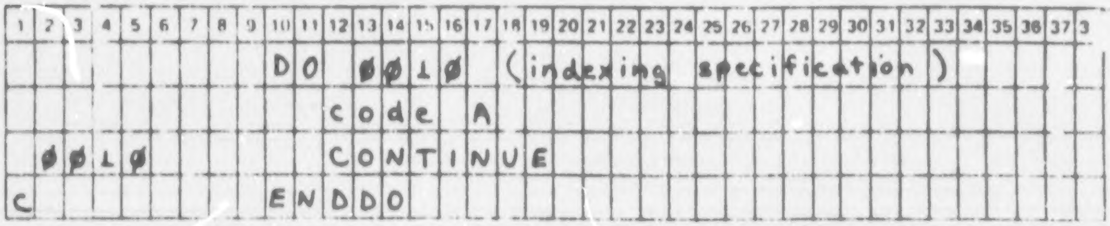
```
C        DO UNTIL (p)
 0010      code A
           IF (.NOT.(p)) GO TO 0010
C        ENDDO
```

The logical expression (p) on the DOUNTIL comment line is recommended but could be deleted at the user's option.

Statements within the figure should be indented two columns from the DOUNTIL and ENDDO comment lines which aid in locating the beginning and end of the figure.

The FORTRAN DO

The FORTRAN DO statement is a command to execute, at least once, the statements that physically follow it, up to and including the numbered statement indicating the end of the DO's range. Since the FORTRAN DO is essentially a specialized DOUNTIL figure, it is desirable that specific guidelines be stated for using this FORTRAN capability. The code structure recommended when using the FORTRAN DO statement is:



```
         DO 0010 (indexing specification)
           code A
 0010      CONTINUE
C        ENDDO
```

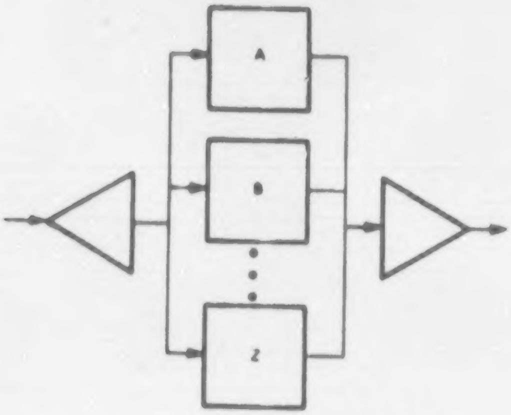
When appropriate, the CONTINUE can be omitted with the statement number defining the end of the range of the DO placed on the last executable statement in the range of the DO.

The statements within this specialized DOUNTIL figure should be indented two columns from the DO statement and the ENDDO comment line which aids in determining the end of the figure.

CASE Figure

The CASE figure causes control to be passed to one of a set of functional blocks of code (A,B,...,Z) based on the value of an integer variable i, equal to (1,2,...,m). It is recommended that the CASE figure be simulated in

FORTRAN using a computed GOTO statement, GOTO statements and a single collector (CONTINUE statement) at the end of the figure. The flowchart for the CASE figure is:



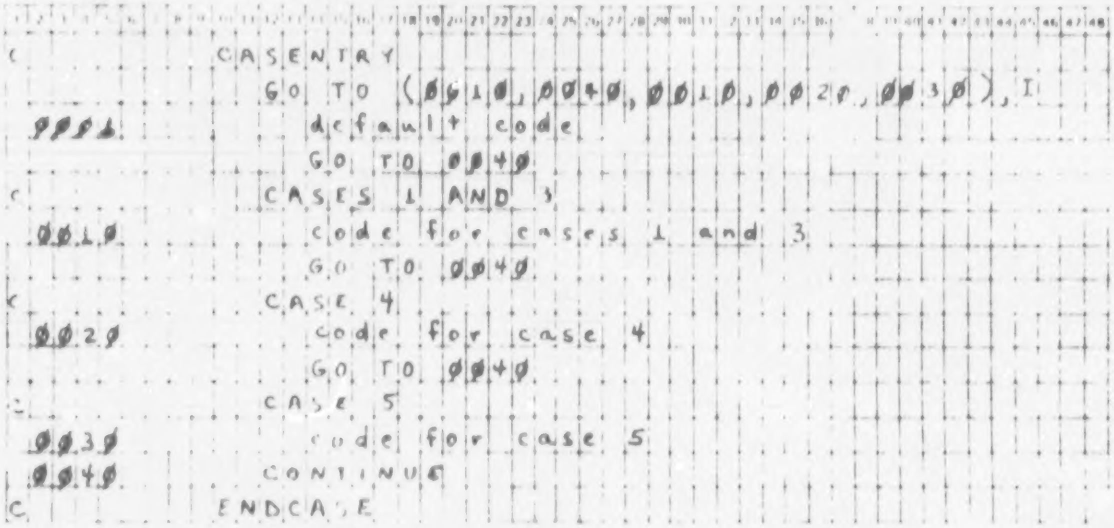
CASE

Some FORTRAN compilers provide that if the value of i is outside the range $1 \le i \le m$, the next statement is executed. However with ANS FORTRAN the computed GOTO statement is undefined if it is not within the range. Therefore in ANS FORTRAN, i should be tested prior to entering the CASE control logic structure. Out of range values should result in control being transfered to the default code.

The default code and the "GOTO 00n0" immediately following the computed GO statement are provided for use with compilers that provide for execution of the next sequential statement when i is not within the range of the computed GOTO.

Statements within the figure are indented two columns from the CASENTRY and ENDCASE comment lines which aid in determining the beginning and end of the figure. Statements within each case are indented two columns from each CASE comment line.

If the functional blocks of code are identical for more than one case, the appropriate entries in the computed GOTO statement should contain the same number. Further, if no action is to be performed for specific values of i, the appropriate entries should point to the end of the figure. Consider the following example:



```
C         CASENTRY
            GO TO (0010,0040,0010,0020,0030),I
 0001         default code
              GO TO 0040
C           CASES 1 AND 3
 0010         code for cases 1 and 3
              GO TO 0040
C           CASE 4
 0020         code for case 4
              GO TO 0040
C           CASE 5
 0030         code for case 5
 0040       CONTINUE
C         ENDCASE
```

For i=1 and i=3, the same functional block of code will be executed and for i=2, no processing will be performed.

INCLUDE Capability

The capability of nesting blocks of code within other code blocks is a necessity for top down programming. This is most easily achieved if the language has a compiler directing instruction such as INCLUDE or COPY. In the case of ANS FORTRAN this type of statement does not exist and therefore the effect of nesting may be simulated by the use of nested CALLs of subroutines. However, since the linkages generated by CALL statements may be costly in terms of overhead, two other standard alternative simulations of the nested INCLUDE capability are presented. The first may be used if the included code segment appears in only one place in the program. It is written as follows:

```
C        INCLUDE function name
           GO TO 0010
 0020      CONTINUE
C        END INCLUDE
```

The function name on the INCLUDE comment should be meaningful enough to indicate the processing performed by the out-of-line code. The out-of-line code then terminates with an explicit GOTO to the CONTINUE statement. If the same functional process is to be simulated as an INCLUDE in more than one place in the program, the assigned GOTO may be used to return control from the out-of-line code as follows:

```
C        INCLUDE function name
           ASSIGN 0010 TO I
           GO TO 1000
 0010      CONTINUE
C        END INCLUDE
         .
         .
         .
C        INCLUDE function name
           ASSIGN 0020 TO I
           GO TO 1000
 0020      CONTINUE
C        END INCLUDE
         .
         .
         .
```

The code which begins at statement 1000 when terminated with the following assigned GOTO statement:

```
           GO TO I, (0010,0020)
```

then returns control to the correct point to complete the simulation of the nested INCLUDE.

The major problem encountered with the lack of an INCLUDE directive is related to the adverse effect this has on both the debugging procedures and the office and machine procedures in the program support library. Ideally the most current listing of a block of source code is filed in a notebook where it may be examined by any person who wishes to do so. This implies a mechanism for storing such blocks as individual entities in a data set which has a directory of names that permits selective access of these small entities. A change can be made to any block without the necessity of passing all existing code through the editing routine and the filing of the revised listing in the library does not require replacement of other module listings. However, the input to the FORTRAN compiler requires that all of these individual subroutine blocks be gathered into a single sequential data set before being fed into the compiler. It is this capability which is supplied with the nested INCLUDE or COPY and the lack of it means that the program must be developed as a single sequential data set. In order to handle this problem various solutions outside the scope of the language have been implemented, such as precompilers, linkage editor INCLUDEs, or a data set concatenation capability within the operating system.

Additional Recommended Coding Conventions

Restricted FORTRAN Statement Usage

In order to maintain structured programming concepts, it is recommended that certain allowable ANS FORTRAN statements generally not be used except as required in the previous definition of the standard program figures and summarized below. For the most part, an attempt is made to preclude unconditional branching not necessitated by standard program figure definition.

The GOTO statement is used in the definition of the following standard program figures: IFTHENELSE, DOWHILE, DOUNTIL and CASE. The computed GOTO statement is used in the definition of the CASE standard program figure. It should be an objective not to use these statements except in those figures.

The ASSIGN and ASSIGNED GOTO statements provide an unconditional branching capability. The arithmetic IF statement is not necessary because the IFTHENELSE standard program figure, with nesting sometimes required, will provide the same capability. Use of these FORTRAN statements should be avoided.

The recommended use of the DO statement as a specialized DOUNTIL, is covered in a previous subsection. Other usage of the DO is not recommended.

The CONTINUE statement is used in the definition of the IFTHENELSE and CASE standard program figures. In addition, it is sometimes required by a DO (specialized DOUNTIL) statement. No other use of the CONTINUE should be necessary.

Program Organization

These conventions provide for the organization of a FORTRAN source program into a set of segments for compilation. Any FORTRAN program requires a certain ordering of the statements within the program. A suggested further restriction to that ordering for the sake of readability, clarity, and consistency appears below.

a. If this is a subprogram, the first card must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

b. Any COMMON statements, each followed by all type, DOUBLE PRECISION, and EQUIVALENCE statements related to it follow. No dimension information is to appear on a COMMON statement. The COMMON statement will be used only to declare the order of arrays and variables within the COMMON. Blank COMMON is to be declared first, followed by all labeled COMMONs in alphabetical order.

   Any explicit specification (type) statements and DOUBLE PRECISION statements will be arranged in alphabetical order of the variables or arrays within each of the types. They will be defined in the following order: COMPLEX, DOUBLE PRECISION, REAL, INTEGER, and LOGICAL. All dimensioning information should be included on the type or DOUBLE PRECISION cards. All variables or arrays should be explicitly declared, and the DIMENSION statement should not be used in place of a type statement.

   Following each type or DOUBLE PRECISION statement, any EQUIVALENCE statements required for that type statement are included. A blank comment card should be used before and after the EQUIVALENCE statements to set them off from the surrounding definitions.

c. Once all COMMON declarations are made, the program local declarations are made using the same conventions.

d. Following all program local declarations, all EXTERNAL declarations will be made.

e. Any DATA statements for program local arrays and variables follow.

f. All FORMAT statements follow.

g. Any statement function definitions come next and complete the non-executable code.

h. Segments containing executable code follow in order. The last segment must contain an END card.

i. If desired, subprograms may follow as part of a multiple compilation. The organization of each subprogram should follow the rules given above.

Comments

Comments should be used to enhance the readability and understanding of a program (e.g., to define variables or their special settings). In general, when they are used they should be grouped together as a prologue to the code segment. If they must be interspersed within the code, they should be inserted as a block which begins in a column near the middle of the page (e.g., column 35 or 40) so as not to interfere with the indentation and readability of the program proper which may be scanned near the left margin. Blank comment cards should be used when they enhance readability.

Statement Numbering

As much as possible, statement numbers are to proceed from lowest to highest as a program is read. It is recommended that statement numbers be four digits long, be placed in columns 2-5, and be incremented by 10 rather than be consecutive.

Continuation Cards

ANS FORTRAN permits up to 19 successive continuation cards per statement. The continuation column should be used to indicate the order of the cards. This may be done by placing a numeric character in column 6 (the continuation indication column) in ascending sequence (i.e., 1-9) and if additional characters are necessary, using in order the alphabetic characters A-J.

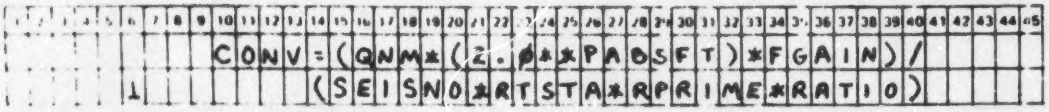
The body of the continuation card should be coded so as to enhance the readability of the program unit. In the following subsections are some suggestions in regard to special cards, but, in general, no continuation card should contain information to the left of the statement identifier on the first card.

Statements

Each statement should begin in an even-numbered column. Non-executable statements should begin in column 8. If the program is a subprogram, the FUNCTION, SUBROUTINE, or BLOCK DATA statement should begin in column 8, and the corresponding END statement should also begin in column 8; in these cases, the first executable statement will begin in column 10. Otherwise, the first executable statement of a program should begin in column 8, as should the END card. Succeeding executable statements are indented according to the coding examples in the standard program figures section and the rules given below.

Assignment Statements

If a statement is continued, the second and following lines should be indented by six columns. For example:



```
         CONV=(QNM*(2.0**PABSFT)*FGAIN)/
     1         (SEISNO*RTSTA*RPRIME*RATIO)
```

In a continued assignment statement, the first card (and each card which will be followed by another continuation card) should have an operator as the last character. As much as possible, all parenthetical expressions should be on a single card. In no case should a left parenthesis be the last character on a card which is to be continued.

COMMON Statements

The COMMON card will begin in column 8 with the identifier followed by a blank in column 14 and a "/" in column 15. The next six columns are reserved for the label which will be left justified in this field. Column 22 will contain another "/" and column 23 will contain a blank. If blank COMMON is desired, code the slashes but leave the label field blank. Columns 24 through 50 will contain the names separated by commas. Commas will appear in columns 30, 37, and 44. If a continuation card is necessary, a comma will appear in column 23 on that card. The names will be left justified within each field.

Type Statements

Dimension information for each array is to be contained on a single type card. A card will contain one and only one name. A type statement may be continued, thus allowing 20 names to be declared for each statement.

The type statement will be coded beginning in column 8. The name should begin in column 16. If it is a continuation card, a comma should be inserted into column 15. When using a DOUBLE PRECISION statement, the name should begin in column 25. If a continuation card is required, a comma should be inserted into column 24.

FORMAT Statements

The FORMAT identifier will be coded beginning in column 8. The first parenthesis will appear in column 16, and the format information itself begins in column 18. As much as possible, a position code and its associated format code should stand alone on a card. Continuation cards should be used liberally. For example:

```
 8060  FORMAT  ( 1H0,20X,8HMONTH = ,I2,
     1                4X,6HDAY = ,  I2,
     2                4X,7HYEAR = , I4)
```

READ or WRITE Statements

In all cases a READ or WRITE statement should have its subject data set reference number (DSRN) contained in a variable. The use of hard coded DSRNs is discouraged from the standpoint of visibility and parameterized coding. The DSRN variables may be loaded with a DATA statement.

The list portion of the READ or WRITE statement must be expressed in as simple terms as possible. Liberal use of blanks and continuation cards is encouraged in order to increase readability.

IF Statements

Multiple conditions in the predicate of an IF statement should occur on separate cards, with an operator occuring as the final item in each card to be continued. For example:

```
         IF (A.GT.B .OR.
     1       C.LT.D .AND.
     2       X.EQ.J
     3      ) GO TO 0100
```

Notice also that the GOTO statement follows the closing parenthesis.

DATA Statements

Only one variable may be specified on a DATA statement card. The DATA identifier will be coded beginning in column 8. The variable name will be coded beginning in column 14. The slash indicating the beginning of the data will be coded in column 20. For example:

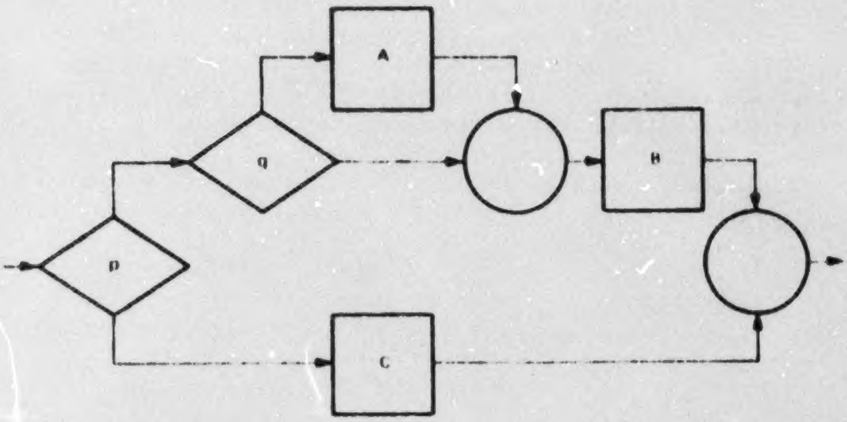
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
							D	A	T	A			Y	E	A	R			/	1	9	7	5	/					

ANS COBOL

General Comments

The two statements IF....ELSE and PERFORM, which are part of the ANS COBOL supply the basic structuring capability of the language. The first represents the IFTHENELSE control figure while the PERFORM permits the looping required by the DOWHILE/DOUNTIL construct. In addition, the GO TO....DEPENDING ON is readily adapted to the simulation of the CASE statement, the COPY assists in top down programming, and the language's SEARCH statement recognizes the potential utility of such a control logic structure. However, even with all of these features, the language does have certain deficiencies in the implementation of top down structured programming technology.

For instance, because of the way the period delimiter affects the syntax of the language, it is not possible to implement the following flowchart using only the COBOL IF statement without duplicating the sequence of statements in code B or duplicating the test on (p):



With the use of a specific delimiter such as an ENDIF, the problems encountered in implementing the above flowchart could be overcome.

The looping capability is achieved in COBOL with the PERFORM statement. All such loops are of the DOWHILE type since the looping condition is tested prior to execution of the code within the loop. This statement permits repetitive code execution under the following conditions. First, the programmer can indicate that the loop is to be executed a specified number of times. Second, an indexing type of loop can be requested with the PERFORM....VARYING option and finally the PERFORM....UNTIL option exists which is equivalent to the structured programming DOWHILE control figure. The selection of the word

UNTIL to indicate loop termination conditions in COBOL is an unfortunate one since in structured programming terminology not only does it imply a different flowchart but the semantics of the looping control is the negative of the structured programming DOWHILE. Thus (ignoring temporarily the flowchart differences between WHILE and UNTIL), the statement "perform a loop WHILE something is true" translates semantically to "perform a loop UNTIL something is not true." Two options are open to the COBOL programmer. One is to simulate the DOWHILE by negating the conditional logic as follows:

PERFORM....UNTIL (NOT p).

The second is to continue to think of the looping condition in the UNTIL logic to which the COBOL programmer is accustomed but to insure that it is understood that the use of the word "UNTIL" does not guarantee execution of the PERFORMed code at least once. For the experienced COBOL programmer it is probable that the latter course is the one which is less error prone.

The possibility of simulating DO loops through use of the IF and GO TO statements as was done for FORTRAN is not recommended for ANS COBOL because of the mechanism which this language uses to implement comments. An example of what the DOWHILE would look like, if implemented in this manner is as follows:

```
           NOTE DO UNTIL (P).
             GO TO IF-PARAGRAPH.
       RETURN-POINT.
             code A.
       IF-PARAGRAPH.
             IF NOT (P)
             GO TO RETURN-POINT.
           NOTE ENDDO.
```

As may be observed, the readability of the code is diminished by the paragraph names and the span of the loop is obscured by the word "NOTE". In addition, if a paragraph name immediately precedes the first NOTE statement, the code following it becomes a NOTE paragraph rather than a NOTE statement. Thus, the sentence "GO TO IF-PARAGRAPH." is treated as a comment rather than an imperative statement and the control figure becomes a DOUNTIL rather than a DOWHILE. An extension to ANS COBOL allows comments to be introduced by identifying such statements with an asterisk in column 7 of the card. Should this convention ever be adopted as a part of ANS COBOL it would be worthwhile reexamining the possibility of simulating DO loops and delimiting them with comments.

The COBOL DECLARATIVES SECTION, if present, must appear at the start of the PROCEDURE DIVISION. The code in this section is invoked asynchronously by certain conditions which cannot normally be tested by the programmer. These conditions include input/output label handling procedures, input/output error checking procedures, and report writing procedures. Since these blocks of code are out-of-line, they involve a transfer of control which is invisible to the programmer. Such interruptions of sequential control are usually undersirable in structured programming. However, because of the utility of the DECLARATIVES SECTION, no attempt has been made to restrict its usage, particularly since ANS COBOL requires that control automatically return to the statement following the one which caused the asynchronous interrupt.

The COPY and PERFORM statements, in spite of their limitations, are helpful in implementing top down programming. It is essential in top down programming to have the capability of nesting relatively small blocks of code within other such blocks. However, the COPY compiler directive cannot fulfill this function because it is limited to a single level (i.e., code which is copied cannot have a COPY statement within it). Therefore, it is necessary to simulate this requirement with PERFORM statements since these can be nested. One method of simulating the required nesting is not to permit a COPY statement in any segment which is invoked by the PERFORM statement. Then, after the top module is coded, COPY statements may be used to direct the compiler to include in its compilation the various PERFORMed paragraphs and thus overcome the limitations on the nesting of COPY statements. It should be noted that this type of organization implies that the modules which are copied, are located in a library system of some sort.

Finally ANS COBOL as with many higher level languages, has a free format syntax. This permits the programmer to write statements in a continuous prose format instead of requiring the more desirable format of each new statement starting a new line of code and thus enhancing the readability of the structured code. Finally, in the examples which follow, the parentheses which enclose the conditional expressions are optional.

In summary, the deficiencies of ANS COBOL which affect structured programming are:

a. The limitation of the IF....ELSE statement which restricts the nesting capability.

b. The opposite meaning of PERFORM...UNTIL in ANS COBOL as opposed to the structured programming DOUNTIL.

c. The lack of a DOUNTIL capability.

d. The free form of the language.

e. The lack of specific delimiters such as ENDIF.

f. The inability to place the repetitive code of a looping operation in-line.

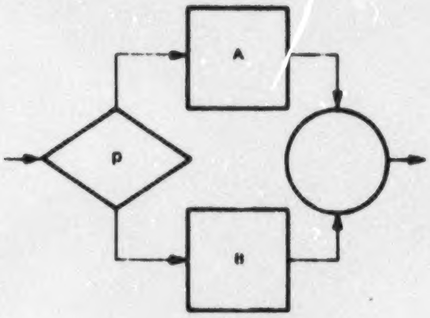
g. The limitation of the COPY statement to a single level and also to the types of COBOL constructs which may be copied.

h. The format of the commenting mechanism which precludes a readable simulation of the looping operation with the IF and GOTO statements.

Top Down Structured Programming in ANS COBOL

IFTHENELSE Figure

The IFTHENELSE figure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p). This is identical to the manner in which the COBOL conditional statement operates so this control logic structure does not have to be simulated. The flowchart for the IFTHENELSE figure is:

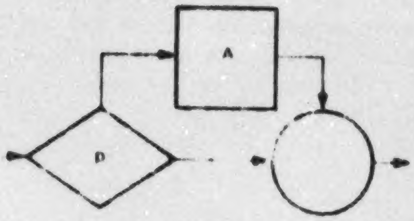


IFTHENELSE

The implementation of this control structure using the conditional statement is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
											I	F		(	p	)													
													c	o	d	e		A											
											E	L	S	E															
													c	o	d	e		B	.										
											c	o	d	e		c	.												

The ELSE in the IFTHENELSE figure is optional and if not used, the flowchart reduces to:



IF THEN

Since the ELSE is also optional in the COBOL conditional statement, the code becomes:

```
           IF (p)
             code A.
           code B.
```

In order to overcome the nesting limitation resulting from the flowchart discussed earlier, the following options are available:

```
           IF (p)
             IF (q)
               code A
               code B
             ELSE
               code B
           ELSE
             code C.
           code D.
```

(a) Duplicate code B

```
           IF (p) AND (q)
             code A.
           IF (p)
             code B
           ELSE
             code C.
           code D.
```

(b) Duplicate the outermost test on (p)

```
           IF (p)
             PERFORM NESTED-IF
           ELSE
             code C.
           code D.
           .
           .
           .
         NESTED-IF.
           IF (q)
             code A.
           code B.
```

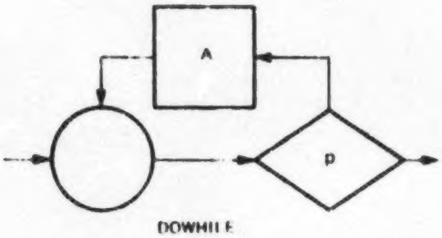
(c) Perform the nested IF statement

The decision as to which option should be adopted as a standard is not a clear cut one and therefore no one form is recommended over any of the others. In the case of the simple flowchart used in the above examples the decision may be based on considerations of space and program execution time. Thus, the duplication of code B is better in terms of time since it is not necessary to execute the compiler generated linkage code of a PERFORM statement or the duplicate test but may be worse in terms of space requirements because of the duplication of code B. However, the above example is an extremely simple one. When these types of control structures occur more than once in more deeply nested code, none of the options avoid the difficulty of becoming considerably more complex. Therefore, each case must be evaluated on its own and the one selected should be the one which is the most readable and understandable. Indentation should be as indicated in the examples. As may be noted, code is indented beneath the IF and ELSE in order to emphasize the span of control of these verbs. Since no unique terminator is available, the end of the figure is indicated by the start of a new code block in the same column as the IF and ELSE.

DO Figures

DOWHILE

The flowchart for the DOWHILE control structure, which is a looping operation in which the test for loop termination precedes the code block, is as follows:



DOWHILE

All of the loops which are initiated by the PERFORM are of the structured programming DOWHILE format and consequently no simulation is needed. The statement is written as:

```
           PERFORM PARAGRAPH-NAME UNTIL (P).
```

The conditional logic of the DOWHILE requires that the loop be terminated on a "false" condition so that in order to be compatable with structured programming definition, the test should read UNTIL (NOT p). However, since the flowchart of the PERFORM statement is that of a DOWHILE, the artificial negation of the termination logic in order to override the semantics of the word UNTIL is not necessary and if (p) were a complex condition, this negation could, in fact, be confusing.

The DOWHILE with indexing is also a part of the language and is written as:

```
           PERFORM PARAGRAPH-NAME
               VARYING ID-1 FROM ID-2 BY ID-3
             UNTIL (P).
```

130

Note that while the indexing is varied automatically, the decision as to when to terminate the loop is still under control of the UNTIL rather than whether the indexing has reached some limit.

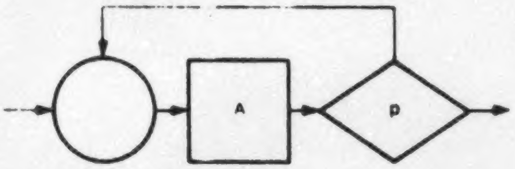
Finally, the option exists to PERFORM a block of code a specified number of times. This is written as:

```
           PERFORM PARAGRAPH-NAME n TIMES.
```

Since, for all PERFORMS, the repetitive code must be placed out-of-line, there are no indentation requirements for these statements. However, for the indexed PERFORM (which can be written with up to three indexing variables), it is suggested that the format of the example shown in this subsection be adopted where words which control the indexing are indented further to the right than the UNTIL so that the conditions for loop termination are emphasised.

DOUNTIL

The DOUNTIL control structure is one in which the looping criteria are tested at the end of the loop and thus the code block is always executed at least once. The flowchart is as follows:



DOUNTIL

Since all PERFORMs are of the DOWHILE control logic, the DOUNTIL must be simulated. The recommended format is either:

```
           PERFORM PARAGRAPH-NAME.
           PERFORM PARAGRAPH-NAME UNTIL (p).
```

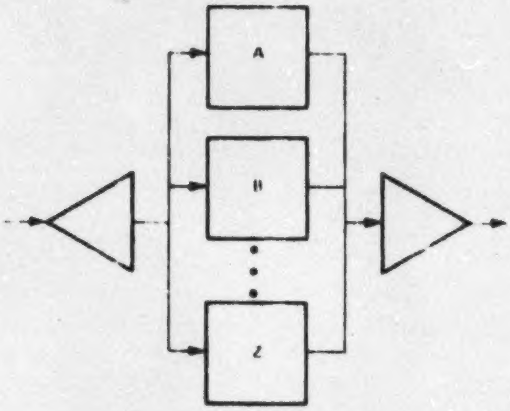
or

```
           TURN OFF LOOP-SWITCH.
           PERFORM PARAGRAPH-NAME
               UNTIL LOOP-SWITCH IS ON.
```

The latter case is most useful with a PERFORM that uses the VARYING option.

CASE Figure

The CASE figure causes control to be passed to one of a set of functional blocks of code (A,B,...,Z) based on the value of an integer variable. The flowchart for this figure is:



CASE

The CASE statement is not part of conventional COBOL and must therefore be simulated using the GO TO...DEPENDING ON statement. This verb permits the programmer to select one of a set of procedures depending upon the value of an integer whose range is from 1 to the number of procedure names listed in the statement. For any integer outside these limits the GO TO statement is ignored and control passes to the statement which follows it. This means that default code, if required, should immediately follow the GO TO...DEPENDING ON statement. At least one paragraph name is required. The simulation is as follows:

```
     CASE-PARAGRAPH.
       GO TO CASE-1 CASE-2 ... DEPENDING ON I.
         default code.
         GO TO CASE-PARAGRAPH-END.
      CASE-1.
         code A.
         GO TO CASE-PARAGRAPH-END.
      .
      .
      .
      .
      CASE-n.
         code Z.
     CASE-PARAGRAPH-END.
       EXIT.
```

The above code must be placed out-of-line and is invoked by an in-line PERFORM statement as shown below:

```
 PERFORM CASE-PARAGRAPH THRU CASE-PARAGRAPH-END.
```

Note that the PERFORM...THRU option of the PERFORM statement must be used because of the presence of the paragraph names which are used to identify the various case statements.

INCLUDE Capability

The capability of nesting blocks of code within other code blocks is a necessity for top down programming. This is best done when such blocks of code are stored and can be accessed by the ANS COBOL COPY verb as separate members on a direct access device in a library system. However, it should be noted that this requirement is a compiler dependency and may not be possible for some ANS compilers. The COPY verb may be used in all divisions except the IDENTIFICATION DIVISION. The discussion which follows is directed toward its use in the PROCEDURE DIVISION. Since the COPY does not permit nesting, it is necessary to simulate this requirement with the use of nested PERFORM statements. The blocks of code which are PERFORMed are presumably stored as separate members which are easily accessed on a direct access device and are referenced for the COBOL compiler by means of COPY statements. This means that no COPY may appear in any block of code which is invoked by a copied PERFORM. With this technique the top level of the PROCEDURE DIVISION looks as follows:

```
       PROCEDURE DIVISION.
         TOP PARAGRAPH.
           code A.
           PERFORM NESTED-PARAGRAPH-1.
           code B.
           STOP RUN.
         NESTED-PARAGRAPH-1.
           COPY LIBRARY-NAME-1.
         NESTED-PARAGRAPH-2.
           COPY LIBRARY-NAME-2.
```

Note that the COPY statement references library names, not paragraph names. "NESTED-PARAGRAPH-1" is a separate block of code which the COPY statement can access and may take the following form:

```
           code C.
           PERFORM NESTED-PARAGRAPH-2.
           code D.
```

"NESTED-PARAGRAPH-2" is a sequence of statements similar to those contained in the above paragraph within which it was invoked and it may contain other PERFORMs for deeper nesting. The COPY statements following the top paragraph insure that the compiler is aware of all the segments of code which comprise the total program. Furthermore, since no PERFORMed paragraph may contain a COPY, there is no danger of violating the nesting limitation of this verb.

Additional Recommended Coding Conventions

Restricted ANS COBOL Statement Usage

In order to preserve the concept of structured programming, it is recommended that the general usage of those statements in COBOL which permit changes of sequential control be restricted to an exception basis only, unless such statements are indicated in the standards as a simulation requirement for the basic control logic figures. Thus, in addition to the GO TO, the ALTER statement should also be limited in its usage.

692

Program Organization

The structure of a COBOL program is such that many of the rules for program organization have been predefined. For instance, all data must be specified in the DATA DIVISION. Furthermore, within this section, the formal rules which define the permissible hierarchical data structures are sufficient to preserve the readability requirements of structured programming. However, within the PROCEDURE DIVISION, (with the exception of the DECLARATIVE SECTION) the rules of COBOL permit the ordering of the PERFORMed code blocks to be completely flexible.

If the program is being developed with the aid of a library system, the order in this division is less critical since all that appears after the top most segment are COPY statements. The functions which exist in the copied code and the functions which are nested within them are determined by examining the small code segments which are present as printed listings of members in the source code library rather than on the compiler output listing even though it is still true that the resolution of the COPY statements by the compiler will produce a complete source program as one of the compiler outputs.

However, for a development process in which no random access library exists, the ordering of the segments of PERFORMed COBOL paragraphs in the procedure division is more critical. This is because the source listing under this condition is a single sequential data set. At present, the suggested sequence is initially by nested level for 2 or 3 levels (depending on the program's complexity) and alphabetically thereafter.

PERFORMed paragraphs should be separated from the main body of code, and from other PERFORMed paragraphs, by at least two blank lines. Logically noncontiguous paragraphs (other than those used in the CASE figure) should be separated by at least one blank line.

Comments

One of the primary intents of the developers of the COBOL language was to produce a self-documenting language. When this is coupled with the discipline of structured programming the resulting programs should be even more readable. Experience has indicated that well written COBOL programs contribute toward meeting this objective. Therefore, it is recommended that the use of comments in the form of NOTE sentences and NOTE paragraphs be held to a minimum. When they are used, they should be organized in such a manner as not to interfere with the readability of the program itself. This may be done by such devices as using blank lines to insure that the NOTE text stands apart from the program proper and starting and concentrating the textual commentary in the middle of the pages beginning in Column 35 or 40.

687

13½

Variables and structures defined in the DATA DIVISION should be arranged in a meaningful order. This order could be alphabetic, by class such as the days of the week, or any other class format. A suggested set of indentation rules for data items is as follows:

> **General Format. All level 77 and 01 variables should have their level numbers in columns 10-11 and names starting in column 14. The PICTURE clause should between columns 32-45, depending on the length of the longest variable name. All other clauses used should follow the PICTURE clause with normal spacing. If more than one line is needed for a variable's definition, the second and succeeding lines should be indented from the PICTURE clause as follows:**

```
  77  RECORD-COUNT      PICTURE 9(7)V99
                            COMP SYNC.
```

> **Structures. Each successively lower level in a structure should be indented four columns from the next higher level. Level numbers should precede each variable name in the structure on the same line and two columns before it as follows:**

```
  01  EMPLOYEE-RECORD.
      02  NAME.
          03  FIRST     PICTURE X(10).
          03  MIDDLE    PICTURE X.
          03  LAST      PICTURE X(20).
      02  ADDRESS.
          03  STREET    PICTURE X(15).
```

> **When condition-names (level 88 items) are used they should be indented and written with single spacing between words:**

```
          03  TYPE-CODE  PICTURE X.
            88  NEW-TYPE VALUE 'B'.
            88  OLD-TYPE VALUE 'Z'.
```

he indentation recommendations for the structured programming figures which appear in the PROCEDURE DIVISION were previously specified in the section where they were described. When programming the code blocks which represent the SEQUENCE in structured programming, each COBOL statement should start a new line. Any statement not subject to indentation rules starts in the same column in which the statement above it started. Only one statement per line is permitted. If any statement must be continued on the next line, the continued portion should be indented 4 to 6 columns so as not to obscure the COBOL verb as follows:

```
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58
  MOVE HOUR TO LAST-HOUR.
  ADD 1 TO HOUR.
  MULTIPLY MEASUREMENTS (HOUR) BY RATE-OF-DECAY
      GIVING RESIDUAL (HOUR).
  MOVE NEW-RATE TO RATE-OF-DECAY.
```